«Системні технології» 2 (163) 2026 «System technologies»
DOI 10.34185/1562-9945-2-163-2026-10
UDC 004.4`4

P.P. Forkert, M.G. Ivanchenko

IMPLEMENTING EXTENSION METHODS AND GENERIC METHODS IN GO PROGRAMMING LANGUAGE DIALECT

Abstract. This paper explores the implementation of extension methods in GoNext, a dialect of the Go programming language that transpiles into standard Go code. Go is a statically compiled language with a lightweight runtime, excellent concurrency support, fast compilation, and a large ecosystem, but it also has a relatively simple type system that does not allow defining methods on types outside the package where they are declared. Extension methods address this limitation by allowing developers to add methods to existing types – including those defined in third-party libraries or the Go standard library – without modifying their source code. The authors review how extension methods are implemented in other mainstream programming languages (C#, Swift, Kotlin, Rust, Scala) and compare them with the related concept of uniform function call syntax, then propose a GoNext-specific approach: extension functions are declared using a new extension keyword before a regular function declaration and are transpiled into ordinary free functions by simply removing that keyword, while call sites are rewritten from method-call syntax (value.Method(params)) into explicit function calls (package.Method(value, params)) using a resolution algorithm based on Go's type unification rules. The paper also describes an import extension modifier that allows treating all functions in an existing Go package as extension methods without adding unnecessary wrappers. The proposed approach is demonstrated on a complete example showing how chained extension method calls are transformed into nested function calls during transpilation. The paper demonstrates that this approach achieves zero runtime overhead, maintains full two-way interoperability with standard Go, and – as an additional benefit – enables support for generic methods, a feature intentionally omitted from Go due to unresolved questions around existing interfaces implementation. The proposed mechanism leverages transpilation as a practical strategy for extending a language's capabilities while reusing its existing compiler and runtime infrastructure.
Keywords: Go, compilation, transpilation, programming, programming language, method, function, extension, information technology, algorithm.

Introduction. There is a wide range of programming languages suitable for various use cases, each making different trade-offs to achieve its goals. And over time, their number only keeps growing. Most of them never become popular, but the ones that do usually offer specific features that are either absent in other programming languages, are not as convenient to use, or do not integrate well into the language ecosystem. Quite a few of the recently created programming languages are not built from scratch but instead utilize existing tools that provide

© Forkert P.P., Ivanchenko M.G., 2026
Copyright for this paper by its authors. Use permitted under License CC BY 4.0.
ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

low-level building blocks, which aid in bootstrapping and maintaining a language implementation, such as:

- JVM [1], which was initially designed to run and JIT-compile Java but has since expanded its feature set to support a wide range of different programming languages, both statically typed (Scala, Kotlin) and dynamically typed (JRuby, Jython, Clojure).
- LLVM [2], which revolutionized the programming language compilers space by introducing LLVM IR - a low-level, generic representation of code that LLVM understands, can optimize, and compile into an executable file for a particular operating system and CPU architecture. So, different language implementations that target LLVM need to implement the transformation of source code into the IR and get many optimization passes and machine code generation basically for free. LLVM features both implementations of previously existing languages like C and C++ and newer languages that use its infrastructure like Rust, Zig, or Swift.
- JavaScript VMs. Due to JavaScript being the only language that is fully supported by browsers, there are many languages that transpile [3] into it, so that developers can write applications that run in browsers and not use JavaScript, which might not be ideal for some specific use cases. Notable representatives of such languages are Dart, ClojureScript, TypeScript and so on.

Go [4] is a statically compiled programming language, featuring a lightweight and efficient runtime with an excellent concurrent garbage collector, scheduler, support for concurrency, fast compilation, and a huge ecosystem of existing libraries. It also has quite a rudimentary type system, and programs written in it are usually quite verbose.

Therefore, it was decided to explore [5-6] whether it can itself serve as a low-level building block for implementing more featureful programming languages and what trade-offs its runtime will impose on those language implementations.

One of the ways to check whether the language with some feature can be implemented on top of Go runtime is to take this feature and add it to Go compiler itself (and potentially lead to the feature being implemented in standard Go itself). But since there is no stable interface for the Go runtime, it is not possible to build another compiler that targets it directly.

Instead, it is possible to build a Go dialect with additional features [7-8] and transpile it into Go itself, and then let the standard Go compiler transform the resulting code into an executable binary.

Problem statement. It can be convenient to be able to “extend” existing Go types, that are defined in 3rd party libraries or in Go standard library, by adding methods (or at least something that would look like a method) to these types without changing their source code.

For example, there is a type in some library:

```
// package library
type IntArray []int
And a code in the application that uses it:
// package application
import library
import ext
```


ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

```
func use(arr library.IntArray) int {
  return arr.Sum()
}
```

It should be possible to implement package ext so that it adds the Sum method to the IntArray defined in 3rd party library.

Methods like Sum are generally named “extension methods” and this paper describes a way to implement this functionality in a Go dialect called GoNext and transpile it into standard Go code that could then be compiled by the official Go compiler to a binary executable.

Main requirements for such implementation are:

- Minimal runtime overhead. Ideally, the code that such “extension methods” transpile to would execute as fast as the code that would be written by a developer for standard Go.
- Full two-way interoperability with standard Go. Existing libraries written in standard Go should benefit from the “extension methods” feature, if possible, and GoNext libraries with extension methods should be usable from standard Go.

Review of the literature. An extension method is a static method or a free function that, when it is called, looks like an instance method on the extended type. Also, an extension method does not have to be defined in the same place where the type it extends is defined.

Extension methods were first introduced in a mainstream programming language natively in C# 3.0 in 2007 [9]. Defining a new extension method on the type string looks like this (it is a regular static method with an additional this modifier on the first method parameter):

```
public static class MyExtensions
{
    public static int WordCount(this string str) =>
        str.Split(' ').Length;
}
And using it looks like this:
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Extension methods allow adding “not important” (in this example, class string is fully functional without WordCount method, but for some use cases it might be very convenient to use), but convenient methods to different types, which enables building APIs that are easier to use (especially chainable methods on the collection-like types). In addition, modern IDEs offer a better autocomplete experience for such extension methods than for static methods/free functions.

Many modern programming languages support extension methods (or properties) either directly or via adjacent language features that allow developers to implement similar functionality:

- Swift [10] and Kotlin [11] have “extensions” that allow adding methods and computed properties to already defined types. Swift also allows adding protocol conformance via extensions.

- Rust [12] allows us to define a trait and immediately implement it for a type, extending it with the methods specified in the trait, effectively providing the same functionality as extension methods.
- Scala 3 [13] supports extension methods natively, whereas Scala 2 [14] provides the same functionality via implicit classes.

As demonstrated, extension methods are a useful language feature that can be implemented in quite different ways, but all implementations share a couple of important properties:

- Since extension methods are just free functions that take a method receiver as their first parameter, they cannot add fields to it or change its memory layout in any way.
- Extension methods usually do not participate in any late-binding activities (except in Swift, where they can create a witness table for an extension conforming to a protocol).

Another programming language feature that can be used to achieve the functionality of "adding methods to existing types" is uniform function call syntax [15], which was already analyzed in the context of the GoNext language dialect [16].

With a uniform function call syntax implemented in a programming language, effectively every free function automatically becomes an extension method. Basically, uniform function call syntax can be seen as a global thing that affects every method call site, and if some free functions are in the current scope, it's not possible to disable them from being treated as instance methods. On the other hand extension methods are a very local thing from this point of view – only functions that are explicitly marked to be extension methods will be considered like them, which can lead to more obvious behavior when developers try to use some APIs.

When choosing whether a language should implement extension methods or uniform function call syntax from a practical perspective, it is useful to look at existing languages: extension methods feature appears in more languages, and the languages in which it appears are much more popular than the ones that have uniform function call syntax implemented.

Main material. To analyze how extension methods can be implemented in GoNext and transpiled to Go, we need to follow the same four aspects that were described in [7] when implementing full-featured enums in GoNext:

- First, a definition of a feature in the guest language (GoNext) should be presented, in this case: what extension methods definitions look like and how to use them at call sites.
- Translation to the host language (Go): how extension methods themselves and their call sites are transformed into the standard Go code
- Explanation of how a new feature is working with other guest language features, and what effect it has on developers using the language.
- Interoperability concerns: can standard Go code use extension methods directly (it depends on how they are translated to Go), and can GoNext use existing Go libraries as sets of extension methods?

Extension methods definition. Unlike all the mentioned implementations of extension methods in other programming languages, in Go function or method definitions are always

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

defined separately from each other and from the type they are defined on (in case of methods), therefore since GoNext is a dialect of Go and it should feel like “Go with additional features”, it is reasonable to make extension method definitions look similar to regular function definitions. To separate extension methods and regular functions a new extension modifier can be used:

```
extension func Sum(arr library.IntArray) int {
    // ...
}
```

Grammar-wise it can be described as:

```
ExtensionFunctionDecl = "extension" FunctionDecl
```

Where `FunctionDecl` is defined in Go specification [17].

It is possible to express this declaration in a way that requires at least one function parameter inside the grammar, but from an implementation perspective, it is easier to reject extension methods without any parameters after it is parsed.

Extension methods call-site. From the call site perspective, extension methods look the same as calling regular type methods.

Translation of extension method definitions. Extension method definitions can be translated to Go code by simply removing the extension keyword. Thus, they will appear to be just regular methods from a Go perspective.

Translation of extension method call sites. The algorithm can follow the logic described in [16] for unified function call syntax: whenever a call like value.Method(param1, param2) is found, and there is no method Method defined on value’s type, an additional search is started which tries to find an extension method with the matching name and unify-able type (following the Go specification type unification rules [17]) of the first method parameter to the value’s type in the current file’s scope. If such a method is found and there is exactly one such method, the call is replaced with package_name.Method(value, param1, param2). If zero or more than one match is found, it will lead to a regular Go “no field or method” error.

Interaction with other GoNext language features. When a package with extension methods is imported into the current scope, all its extension methods will participate in the extension method search described above. By implementing imports this way, GoNext lets developers control what extension methods are available in the current scope.

Using extension methods from Go. After transpilation, extension method definitions are transformed into regular functions; therefore, they can be used as is in standard Go code. So, from GoNext's perspective, extension methods can be used in two ways: value.Method(params) or package_name.Method(value, params), and only the second way is valid from the perspective of standard Go code.

Using existing Go libraries as if they contained extension methods. Some Go libraries like “lo” contain only free functions that technically can be used as if they were extension methods, but since they already aren’t, they must be wrapped in extension methods to participate into method resolution mechanism described above. For example, for the Filter function from the “lo”, the following “wrapper” needs to be defined:

```
// package loext
```

```
import "github.com/samber/lo"
extension func Filter[T any](
    collection []T,
    predicate func(item T, index int) bool) []T {
    return lo.Filter(collection, predicate)
}
```

And then loext package can be imported and work with extension methods like described above.

Instead of requiring such “wrapper packages”, it is possible to create an additional import modifier, so that in GoNext it is possible to simply do:

```
import extension "github.com/samber/lo"
```

And within the scope of the current file, all functions defined in the package lo will automatically be treated as if they were extension methods.

Combining everything. The easiest way to combine everything described above into a single holistic language feature is to follow all steps of the transpilation process in the relatively simple example.

First, let’s define some extension methods in the sliceext package:

```
package sliceext
extension func Filter[T any](
    collection []T,
    predicate func(item T, index int) bool) []T {
    //
}
extension func Map[T any, R any](
    collection []T,
    fn func(item T, index int) R) []R {
    //
}
```

Then let’s imagine an application code that uses these methods:

```
package application
import (
    "fmt"
    "sliceext"
)
func main() {
    collection := []int{1, 2, 3, 4, 5, 6}
    result := collection.Filter(func(item int, index int) bool {
        return item % 2 == 0;
    }).Map(func(item int, index int) int {
        return item * item;
    }).Filter(func(item int, index int) bool {
        return item > 4;
    })
    fmt.Println(result)
}
```


ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Example contains 3 extension method calls, that are being called in a chain.

After transpilation the sliceext package looks the same but with “extension” modifiers removed:

```
package sliceext
func Filter[T any](
    collection []T,
    predicate func(item T, index int) bool) []T {
    //
}
func Map[T any, R any](
    collection []T,
    fn func(item T, index int) R) []R {
    //
}
```

And the application code, on the other hand, is being converted into this:

```
package application
import (
    "fmt"
    "sliceext"
)
func main() {
    collection := []int{1, 2, 3, 4, 5, 6}
    result := sliceext.Filter(
        sliceext.Map(
            sliceext.Filter(
                collection,
                func(item int, index int) bool {
                    return item % 2 == 0;
                }
            ),
            func(item int, index int) int {
                return item * item;
            }
        ),
        func(item int, index int) bool {
            return item > 4;
        }
    )
    fmt.Println(result)
}
```

So instead of 3 chained calls, it was rewritten into 3 nested calls.

Conclusions. This paper showcases that it is possible to implement a functionality like extension methods on top of Go runtime by using transpilation into the standard Go code.

The requirements described in the problem statement are also satisfied in the proposed implementation:

- The overhead of using extension methods, compared to not using them and instead making function calls explicitly, is zero. In situations where the “wrapper package” for the existing implementation of some functionality is needed, the overhead is a single additional method call per extension. But since the extension methods in this case are one-line wrappers for functions implemented in an existing Go library, the Go compiler will inline them in most cases, so the overhead will effectively be zero too.

- Since extension methods do not affect any language features other than changing method resolution at call sites, and even that is done in a backwards-compatible way because they are applied only when the code would not have compiled at all previously, they do not really affect interoperability between GoNext and Go itself. Go cannot use extension methods as extension methods because it doesn’t know that concept, but since they are transpiled into regular functions, it is possible to just import the package and call them directly from Go. There is also an additional solution that allows importing existing Go libraries as if their functions were written as extension methods, thereby eliminating the need to write “wrapper packages”.

Also, as shown in the example in the section above, the proposed solution provides an additional accidental benefit for the Go programming language – support for generic methods, which Go currently lacks.

Initially, Go did not support generics at all because the team that developed the language believed they would unnecessarily complicate both the language itself and the ecosystem around it. It took a very long time to decide that Go indeed needs generics and to design a way to introduce them to the language [18].

But as described in the Go generics proposal, the generic methods were intentionally omitted from the language [19]. In Go, one of the primary roles of methods is to permit types to implement interfaces. And it is not clear whether it is reasonably possible to permit parameterized methods to implement interfaces. So, it is possible to have methods on generic types, but it is not possible to add new parameterized types to the method definitions.

The Filter method is valid:

```
type Array[T any] struct {
    data []T
}
func (array Array[T]) Filter(
    predicate func(item T, index int) bool) Array[T] {
       data := make([]T, 0, len(array.data))
       for i := range collection {
              if predicate(array.data[i], i) {
                     result = append(result, array.data[i])
              }
       }
       return Array{data: data}
}
```

But the Map method is not, because it requires an additional type parameter R, which is not possible to add anywhere:

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

```
func (array Array[T]) Map(fn func(item T, index int) R) Array[R] {
	data := make([]T, 0, len(array.data))
	for i := range array.data {
		if predicate(array.data[i], i) {
			data = append(data, array.data[i])
		}
	}
	return Array[T]{data: data}
}
```

Extension methods do not interact with interfaces, and they can define any number of type parameters; hence, implementing the Map free function as an extension method is possible even with the current Go compiler.

REFERENCES

1. Wimmer C., Würthinger T. Truffle: a self-optimizing runtime system: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12), 2012. P. 13–14. DOI: 10.1145/2384716.2384723.

2. Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation: Proceedings of the International Symposium on Code Generation and Optimization (CGO'04), 2004. P. 75–86. DOI: 10.1109/CGO.2004.1281665.

3. Bastidas Fuertes A., Pérez M., Meza Hormaza J. Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. Applied Sciences. 2023. V. 13, No. 6. Article 3667. DOI: 10.3390/app13063667.

4. Cox R., Griesemer R., Pike R., Taylor I. L., Thompson K. The Go programming language and environment. Communications of the ACM. 2022. V. 65, No. 5. P. 70–78. DOI: 10.1145/3488716.

5. Forkert P. P., Sydorova M. G. ADVANTAGES OF GOLANG AS A FOUNDATION FOR NEW PROGRAMMING LANGUAGES: Conference materials of the XXI international scientific and practical conference «MATHEMATICS AND SOFTWARE FOR INTELLIGENT SYSTEMS (MPZIS-2023)», Dnipro, November 22-24, 2023. P. 7–8.

6. Forkert P. P., Sydorova M. G. CHALLENGES OF USING GOLANG AS A FOUNDATION FOR NEW PROGRAMMING LANGUAGES: Conference materials of the VI All-Ukrainian scientific and practical internet-conference of young scholars and higher education applicants «Modern informational systems and technologies » on the topic: «Modern computer systems and networks in management », Khmelnytskyi, November 30, 2023. P. 55–56.

7. Forkert P. P., Sydorova M. G. Integrating full-featured enums into Go programming language. Current problems of automation and information technologies. 2023. V.27. P. 3-16. DOI: 10.15421/432301.

8. Forkert P. P., Sydorova M. G. IMPROVING ENUMS IN GO PROGRAMMING LANGUAGE DIALECT: Conference materials of the VI international scientific and practical conference of young scholars and students «SOFTWARE ENGINEERING AND ADVANCED INFORMATION TECHNOLOGIES (SOFT TECH-2024)», Kyiv, May 21-23, 2024. P. 148-150

9. Bierman G. M., Meijer E., Torgersen M. Lost in translation: formalizing proposed extensions to C#: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07), Montreal, 2007. P. 479–498. DOI: 10.1145/1297027.1297063.
10. Extensions | Documentation. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/extensions#Extension-Syntax (date of access: 10.01.2026).
11. Extensions | Kotlin Documentation. URL: https://kotlinlang.org/docs/extensions.html (date of access: 10.01.2026).
12. Defining Shared Behavior with Traits – The Rust Programming Language.
URL: https://doc.rust-lang.org/book/ch10-02-traits.html (date of access: 10.01.2026).
13. Extension Methods.
URL: https://docs.scala-lang.org/scala3/reference/contextual/extension-methods.html (date of access: 10.01.2026).
14. Křikava F., Miller H., Vitek J. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild: Proceedings of the ACM on Programming Languages. 2019. V. 3, OOPSLA. Article 163. DOI: 10.1145/3360589.
15. Bright W., Alexandrescu A., Parker M. Origins of the D programming language: Proceedings of the ACM on Programming Languages. 2020. V. 4, HOPL. Article 73. DOI: 10.1145/3386323.
16. Forkert P.P, Ivanchenko M.G. Uniform function call syntax in Go programming language dialect. Conference materials of the IV international scientific and practical conference «INFORMATION TECHNOLOGIES IN EDUCATION AND SCIENCE», Zaporizhzhia, May 20, 2025. P. 561-565
17. The Go Programming Language Specification. URL: https://go.dev/ref/spec (date of access: 10.01.2026).
18. Griesemer, Robert, et al. Featherweight Go: Proceedings of the ACM on Programming Languages 4. OOPSLA, 2020. C. 1–29. DOI: 10.1145/3428217.
19. Type Parameters Proposal.
URL: https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md#No-parameterized-methods (date of access: 10.01.2026).

Received 11.03.2026
Accepted 13.03.2026
Published 31.03.2026

Реалізація методів розширення та узагальнених методів у діалекті мови програмування Go

У цій статті досліджується реалізація методів розширення в GoNext, діалекті мови програмування Go, який транслюється у стандартний код Go. Go — це статично компільована мова з легким середовищем виконання, чудовою підтримкою паралельності, швидкою компіляцією та великою екосистемою, але вона також має відносно просту систему типів, яка не дозволяє визначати методи для типів поза пакетом, де вони оголошені. Методи розширення усувають це обмеження, дозволяючи розробникам додавати методи до існуючих типів, включаючи ті, що визначені в сторонніх бібліотеках або стандартній бібліотеці Go, без зміни їхнього вихідного коду. Автори

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

розглядають, як методи розширення реалізовані в інших поширених мовах програмування (C#, Swift, Kotlin, Rust, Scala) та порівнюють їх з відповідною концепцією уніфікованого синтаксису виклику функцій, а потім пропонують специфічний для GoNext підхід: функції розширення оголошуються за допомогою нового ключового слова розширення перед оголошенням звичайної функції та транспілюються у звичайні вільні функції шляхом простого видалення цього ключового слова, тоді як сайти викликів переписуються з синтаксису виклику методу (value.Method(params)) у явні виклики функцій (package.Method(value, params)) за допомогою алгоритму розв'язання, заснованого на правилах об'єднання типів Go. У статті також описано модифікатор розширення імпорту, який дозволяє розглядати всі функції в існуючому пакеті Go як методи розширення без додавання зайвих обгорток. Запропонований підхід демонструється на повному прикладі, що показує, як ланцюгові виклики методів розширення перетворюються на вкладені виклики функцій під час транспіляції. У статті демонструється, що цей підхід досягає нульових накладних витрат часу виконання, підтримує повну двосторонню сумісність зі стандартним Go та – як додаткову перевагу – забезпечує підтримку універсальних методів, функцію, навмисно виключену з Go через невирішені питання щодо реалізації існуючих інтерфейсів. Запропонований механізм використовує транспіляцію як практичну стратегію для розширення можливостей мови, одночасно повторно використовуючи її існуючу інфраструктуру компілятора та середовища виконання.

Форкерт Павло Павлович – аспірант кафедри інженерії програмного забезпечення та інформаційних технологій кафедри факультету прикладної математики та інформаційних технологій Дніпропетровського Національного Університету ім. Олеся Гончара.
ORCID: https://orcid.org/0009-0005-7027-1273
Іванченко Марина Генадіївна – к.т.н., доцент кафедри інженерії програмного забезпечення та інформаційних технологій кафедри факультету прикладної математики та інформаційних технологій Дніпропетровського Національного Університету ім. Олеся Гончара.
ORCID: https://orcid.org/0000-0001-7795-0459

Forkert Pavlo Pavlovych – PHD student of the Department of Software Engineering and Information Technologies, Faculty of Applied Mathematics and Information Technologies, Oles Honchar Dnipro National University.
ORCID: https://orcid.org/0009-0005-7027-1273
Ivanchenko Maryna Genadiivna - candidate of engineering sciences, assistant professor of the Department of Software Engineering and Information Technologies, Faculty of Applied Mathematics and Information Technologies, Oles Honchar Dnipro National University.
ORCID: https://orcid.org/0000-0001-7795-0459