Baltic J. Modern Computing, Vol. 13 (2025), No. 1, 67-74
https://doi.org/10.22364/bjmc.2025.13.1.04

Implicit Parameter Scope Handling in Programming Languages

Mikus VANAGS

Logics Research Centre, Sterstu street 7-6, Riga, LV 1004, Latvia

mikus.vanags@logicsresearchcentre.com

ORCID 0009-0001-4542-7097

Abstract: This paper introduces a novel abstract syntax approach designed to simplify the scope and implicit parameter management in nested anonymous methods across programming languages. The proposed innovations include: 1) non-capturing function - a new method for declaring anonymous methods that does not capture implicit parameters, and 2) shorthand higher-order function call - a novel technique for invoking methods that captures implicit parameters within the scope of the function call, thereby generating a new anonymous function to be passed to the calling function. These advancements enable a more concise syntax for anonymous methods, enhancing code readability. Furthermore, the approach to implicit parameter handling in nested anonymous methods improves the conceptual understanding of boundaries and interactions between complex nested anonymous functions. Collectively, these innovations pave the way for more intuitive, maintainable, and expressive anonymous methods in programming languages.

Keywords: programming languages, implicit parameter, anonymous methods, parameter scope.

1. Implicit parameters

Scala language has feature named ‘contextual parameters’ aka ‘implicit parameters’ (WEB, a) which is something between global variables and default arguments rather than feature completely enclosed inside the method body declaration as are method parameters. Kotlin® supports keyword ‘it’ (WEB, b) – it can be used in lambda expressions as single parameter with constant name which might affect code readability. Q language supports up to 3 implicit parameters with special names x, y and z (WEB, c) which also is a limitation of the expressiveness and might affect code readability. Swift® has shorthand argument names (WEB, d) which allows to refer to lambda parameter using index of the parameter - that is similar idea to Clojure shorthand lambda syntax (WEB, e) - to use indexes instead of names which also might affect code readability. Accessing parameters by indexes or using constant names to access the parameters is ambiguous in nested lambda expressions, therefore is needed better, more abstract and more expressive model of implicit parameters (Vanags and Cevere, 2018).

The idea of implicit parameters redefine how parameters are declared in programming, moving the parameter declaration from the method's signature to the body of the method. In this approach, all unknown identifiers within the method body are treated as parameters that have been implicitly declared. These implicit parameters are handled as expressions, and the first occurrence of an implicit parameter expression leads to its addition to the list of method parameters (Vanags et al., 2016). In the following pseudocode example parameter ‘x’ is implicitly defined:

```
function {return x+6;}
```

Following pseudocode demonstrates equivalent example declaring parameter explicitly (the usual way how it is done in programming languages):

```
function(x) {return x+6;}
```

Implicit parameters make method declarations more concise, thereby improving code readability, particularly for lambda expressions (relatively small and simple code expressions), making this approach potentially applicable across various programming languages. Comparison of possible anonymous method syntax improvements related to implicit parameters are shown in Figure 1 demonstrating how implicit parameters facilitate removing unnecessary keywords and symbols from lambda syntax making the lambda syntax very concise.

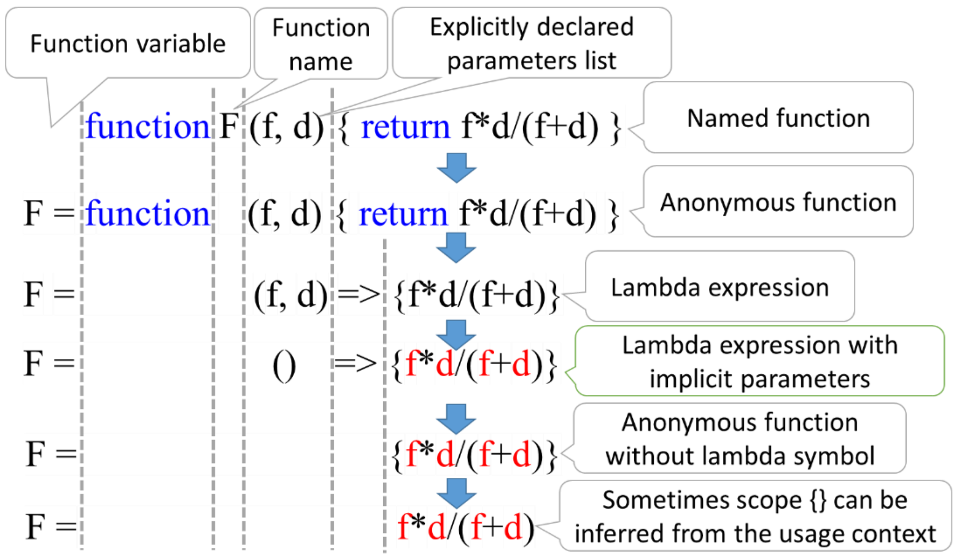

Function variable
Function name
Explicitly declared parameters list
function F (f, d) { return f*d/(f+d) }
Named function
F = function (f, d) { return f*d/(f+d) }
Anonymous function
F = (f, d) => {f*d/(f+d)}
Lambda expression
F = () => {f*d/(f+d)}
Lambda expression with implicit parameters
F = {f*d/(f+d)}
Anonymous function without lambda symbol
F = f*d/(f+d)
Sometimes scope {} can be inferred from the usage context

Figure 1. Anonymous method syntax improvements using implicit parameters.

Implicit parameters are placed in the method's parameter list in the same order as they are identified within the method body. The parameter list is part of the method's signature, and there are occasions when altering the order of these parameters is desirable. Implicit parameter order can be changed using Grace~ operator (Vanags, 2016). The prefix form of the Grace~ operator shifts a parameter one position closer to the beginning of the method's parameter list:

```
function {return y - ~x;}
```

Conversely, the postfix form of the Grace~ operator shifts a parameter by one position towards the end of the method's parameter list:

```
function {return y~ - x;}
```

Both Grace~ operator usage examples are functionally equivalent to the following example without using implicit parameters:

```
function(x, y) {return y - x;}
```

Currently only KatLang programming language (WEB, f) implements implicit parameter feature.

2. Implicit Parameters in Nested Functions

In the following ECMAScript aka JavaScript (WEB, g) example (using explicitly declared parameters) parameters a and b belong to the function assigned to the variable f. In the case of the anonymous function assigned to the function variable g, the parameter c is associated with the nested anonymous function.

```
var f = function(a, b) { return a(b) }
var g = function() {
    return f(function(c) {return c+1}, 2)
}
g()
```

Functionally equivalent example using implicitly declared parameters is as follows:

```
f = function { return a(b) }
g = function {
    return f(function {return c+1}, 2)
}
g()
```

More concise syntax can be achieved by removing all the unnecessary keywords:

```
f = { a(b) }
g = {
    f({ c+1 }, 2)
}
g
```

In a nested functions environment, each implicit parameter is captured by the most nested function within which the implicit parameter is encountered for the first time. The outer anonymous function does not contain any parameters because implicit parameter c is first encountered and thus captured within the nested anonymous function.

It is important to note that an implicit parameter will invariably be captured by some function. If not captured by any inner function, the last opportunity for capture is by the most outer function. Consequently, the most outer function is always defined with {} brackets, but since these brackets are a constant feature of the outermost function, they can be omitted and inferred from the context of the code's use.

Previous example can be improved by removing unnecessary brackets as follows:

```
f = a(b)
g = f({ c+1 }, 2)
g
```

When all the unnecessary symbols and keywords are removed, it is a little easier to notice which anonymous function captures which implicit parameters. The result is KatLang example. KatLang allows for the omission of the outermost function's brackets {} because it interprets line endings as possible ending of an expression. While such a practice may not align with the syntax of all programming languages, it illustrates the potential for making code more concise by eliminating unnecessary constants, symbols, or keywords.

3. Grace~ operator role in changing the scope of implicit parameters

Implicit parameters are limited to the scope in which they are defined, but Grace~ operator can be used to change the scope of an implicit parameter. It means that Grace~ operator prefix form can move implicit parameters to outer scope as demonstrated in following example:

```
f = function{
    return function {
        return ~a+1
    }
}
```

The same example without unnecessary keywords:

```
f = {
    {~a + 1}
}
```

Parameter a is tried to move one position before the beginning of the parameters list and it is interpreted as moving the parameter to the end of the outer function parameters list.

Functionally equivalent JavaScript example using explicitly declared parameters are as follows:

```
var f = function(a) {
    return function() {
        return a + 1
    }
}
```

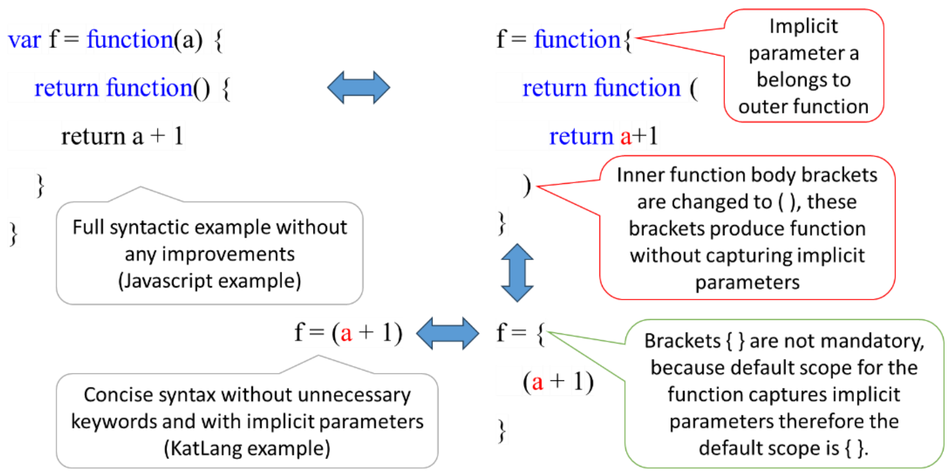
The postfix form of Grace~ operator does not have capability to change the scope of implicit parameter, therefore prefix and postfix forms of the Grace~ operator is asymmetric.

Symmetry, simplicity and code readability are the reasons why Grace~ operator is better to be limited to work only in the scope of single function and do not allow to move the parameter outside the visibility scope of the function. A better solution is needed to control the scope of implicit parameters.

4. Non-capturing function – parameter less function with implicit parameters in the function body

For the sake of simplicity, the function which can contain parameters is called a 'parametrized' function. In the world of implicit parameters, the only way for the function to be parameter less function is not to have any implicit parameter in the body of the function. But that is a serious limitation for the language expressiveness. The limitation can be overcome by creating non-capturing function - a new kind of function which is defined between brackets '(' and ')' and such function does not own or capture any implicit parameter. The implicit parameters are owned by the closest parametrized function and the parametrized function is defined between brackets '{' and '}'. The most outer function is parametrized by default, because some function needs to capture implicit parameters if they are not captured in the inner functions. Therefore, in the definition of the most outer function bracket { } usage is optional.

Figure 2 explains how JavaScript syntax can be improved demonstrating the usage of parameter less function defined withing brackets ( ).



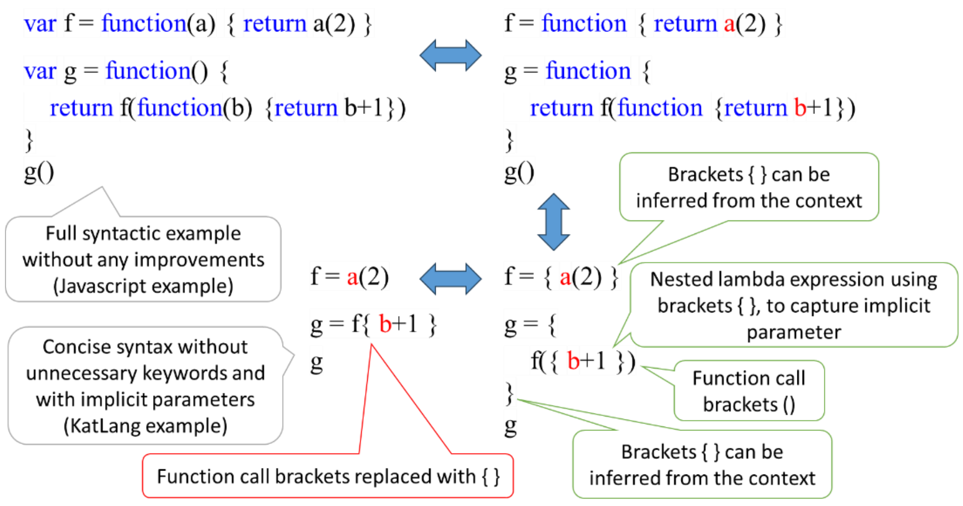
var f = function(a) {
return function() {
return a + 1
}
}
Full syntactic example without any improvements (Javascript example)
f = function{
return function (
return a+1
)
}
Implicit parameter a belongs to outer function
Inner function body brackets are changed to ( ), these brackets produce function without capturing implicit parameters
f = (a + 1)
Concise syntax without unnecessary keywords and with implicit parameters (KatLang example)
f = {
(a + 1)
}
Brackets { } are not mandatory, because default scope for the function captures implicit parameters therefore the default scope is { }.

Figure 2. Deduction of parameter less function syntax with explanations.

Function f=(a+1) contains outer function (which is parametrized by default) and inner function defined with brackets ( ). Usually in programming languages it means that the function can be executed and then the resulting function (returned from the initial call) can be executed. The execution syntax is following: f(1)(). To simplify syntax for such cases when the function is called, the execution can be performed to all the returned (inner) parameter less functions until no more executions are possible. It means unwrapping the nested (inner) parameter less function which results in getting rid of the unnecessary brackets (scopes). Such behavior might not be the best in all possible situations, but for processing math expressions, it makes sense and works well in programming language KatLang.

5. Shorthand higher order function call

If the function body can be declared in two different ways – using brackets () or {} as shown in the previous chapter, then the same principle can be applied to method calls which means implementing the method call using brackets {} instead of (). It means the method call will be specialized to pass lambda expression as parameter to the calling method. Figure 3 shows how to convert explicit parameters example to implicit parameters example, then how to make syntax more concise by removing unnecessary keywords and brackets, and then how to improve syntax by using brackets {} in the method call. This approach simplifies the syntax for shorthand higher-order function calls by allowing a lambda expression to be passed to a function that expects another function as a parameter.



```
var f = function(a) { return a(2) }
var g = function() {
   return f(function(b) {return b+1})
}
g()
```

```
f = function { return a(2) }
g = function {
   return f(function {return b+1})
}
g()
```

```
f = a(2)
g = f{ b+1 }
g
```

```
f = { a(2) }
g = {
   f({ b+1 })
}
g
```

Full syntactic example without any improvements (Javascript example)
Concise syntax without unnecessary keywords and with implicit parameters (KatLang example)
Function call brackets replaced with { }
Brackets { } can be inferred from the context
Nested lambda expression using brackets { }, to capture implicit parameter
Function call brackets ()
Brackets { } can be inferred from the context

Figure 3. Deduction of shorthand higher order function call syntax with explanations.

Bracket {} usage in method call expressions is quite simple – the argument of the method call is a function which body is defined between brackets {}. More interesting now seems bracket () usage in method calls. When brackets () are used for the method call, it can be interpreted as passing parameter less function to the method call. Any parameter less function, including nested parameter less functions, can be unwrapped. Usually programming languages allow passing the final unwrapped expressions of the parameter less function to the method call – it is because in other programming languages brackets () are used to manage the operation priorities without creating a new lambda expression. Parameter less function is like a wrapping layer to the unwrapped expressions and structurally it is similar to the scenario when the parametrized function is passed to the method call.

Parameter less function unwrapping takes processor time, and it might not be the best option for all the programming languages, but nonetheless it is an interesting concept worth considering when designing future programming languages.

6. Summary

This paper introduces two key innovations to improve the handling of implicit parameters in programming languages: the non-capturing function and the shorthand higher-order function call.

The non-capturing function - defined with parentheses () - allows for the inclusion of implicit parameters without capturing them within the function. Instead, implicit parameters are bound to the nearest outer parametrized function, defined with curly brackets {}. This distinction reshapes traditional approaches to function scopes and parameter management, shifting the way developers can think about function definition, parameter binding, and scope visibility.

The Grace~ operator provides a method for reordering implicit parameters within the method's parameter list, but its prefix and postfix forms are asymmetric in controlling parameter scope. While the operator is effective in reordering parameters, it is less ideal for managing the parameter scope in nested functions. The non-capturing function offers a more robust solution to this challenge by separating the logic of the function from parameter ownership.

Additionally, the shorthand higher-order function call simplifies function invocation by allowing the omission of traditional function call brackets () when passing a lambda expression as an argument. This concise syntax enhances readability and reduces unnecessary symbols, particularly in functional programming contexts.

Both innovations have been implemented in the abstract programming language for math calculations - KatLang, which supports implicit parameters and the Grace~ operator. KatLang challenges traditional paradigms of function declaration and usage, encouraging developers to rethink the relationship between a function’s logic and its parameter interface. These contributions represent a conceptual shift that enhances readability, clarity, and expressiveness in programming language design.

References

Vanags, M., Justs, J., Romanovskis, D. (2016). Implicit parameters and implicit arguments in programming languages. US Patent 9361071.

Vanags M. (2016). Grace~ operator for changing order and scope of implicit parameters. US Patent 9417850.

Vanags M., Cevere R. (2018). The Perfect Lambda Syntax. Baltic J. Modern Computing, Vol. 6 (2018), No. 1, 13-30 Retrieved from https://doi.org/10.22364/bjmc.2018.6.1.02

WEB (a). Tour of Scala, Contextual parameters, aka implicit parameters. Retrieved May 5, 2024, from https://docs.scala-lang.org/tour/implicit-parameters.html

WEB (b). Kotlin, Higher-order functions and lambdas (2023). Retrieved May 5, 2024, from: https://kotlinlang.org/docs/lambdas.html#it-implicit-name-of-a-single-parameter

WEB (c). Kdb+ and q documentation, Function notation. Kx Systems, Inc. (2023). Retrieved May 5, 2024, from https://code.kx.com/q/basics/function-notation/

WEB (d). The Swift Programming Language (5.10), Shorthand Argument Names (2023). Apple Inc. Retrieved May 5, 2024, from https://docs.swift.org/swift-book/documentation/the-swift-programming-language/closures/#Shorthand-Argument-Names

WEB (e). Clojure, Reader, Hickey R. (2022). Retrieved May 5, 2024, from https://clojure.org/reference/reader

WEB (f) KatLang. Logics Research Centre (2022). Retrieved May 5, 2024, from http://katlang.org/

WEB (g) ECMAScript® 2023 language specification (2023). Retrieved May 5, 2024, from https://ecma-international.org/publications-and-standards/standards/ecma-262/

Received May 11, 2024, revised September 22, 2024, accepted January 13, 2025