Baltic J. Modern Computing, Vol. 6 (2018), No. 1, 13-30
https://doi.org/10.22364/bjmc.2018.6.1.02

The Perfect Lambda Syntax

Mikus VANAGS, Rudite CEVERE

Department of Computer Systems, Latvia University of Agriculture, Liela iela 2, LV3001, Jelgava, Latvia

mikus.vanags@logicsresearchcentre.com, rudite.cevere@llu.lv

Abstract: Most of the mainstream general-purpose programming languages support lambda expressions. In some languages (JavaScript, Swift) lambda expressions are called "closures". However, this fact does not change their semantic meaning: lambda is an abstraction of mathematical function, which looks like anonymous method or function in programming languages. There exist different perfections in mathematics: perfect numbers, golden ratio, etc. These perfections are characterized by the specific properties. This article presents a review and analysis of a set of characteristics that describe a perfect lambda syntax. Different ideas of lambda expression implementation are compared, and the perfect lambda syntax is discovered.

Keywords: lambda syntax, lambda expression, implicit parameter, Grace~ operator.

1. Introduction

For the first time lambda was introduced in Lambda Calculus (Rojas, 1998) invented by the mathematician Alonzo Church in the thirties of the 20th century and used by the Lisp programming language since 1958 (Veitch, 1998). Lambda is an abstraction of mathematical function and refers to anonymous functions in programming languages (Dushkin, 2006).

```
function (x) { return x * x }; //example of anonymous function
```

Lambda often is used as an expression (something that evaluates to a value); therefore, it is called "lambda expression". Using a lambda instead of a (usual) named function, programmers do not need to worry about the following:

- separating function declaration from function call;
- naming something that is used only once.

Lambda expressions offer a syntactical advantage over a traditional anonymous function declaration by removing a few artefacts like the keywords 'function' and 'return', which makes these expressions feel more like callable blocks of code (Atencio, 2015).

```
x => x * x
```

This syntax makes code more concise when used in conjunction with high-order functions like map, reduce, and filter:

```
range(1, 5).map(x => x * x); //-> [1, 4, 9, 16, 25]
```

Lambdas encourage practicing the first of the famous SOLID principles of software design: Single Responsibility (Martin, 2003). Applied to functions, single responsibility means that functions should have a single purpose. You can have lambda expressions containing multiple statements:

```
(x, y) => {
    // do something with x and y
    // statement 1
    // statement 2
    return result;
}
```

Most of lambda usage derives from being inlined as a function argument into larger expressions or within a composition. This means that you can create several small lambda functions that do exactly one thing, and combine them together to form whole programs:

```
range(1, 5)
    .filter(x => (x % 2) === 0)
    .map(x => x * x);    //-> [4, 16]
```

Lambda expressions encourage creating programs that are declarative, modular, and reusable at a very fine-grained level. However, it turns out, that lambda expressions are not perfect in the sense of this term, as it is dealt with in this article, and lambda syntax can be improved even more. Perfection can be characterized by the specific properties.

Programmers most often read source code in programming language. Therefore, code readability is an important characteristic of the perfect lambda syntax. Programmers also write code and edit code. Code writing is easier if there are fewer symbols to write. The same principle applies to code editing. Code readability depends on the number of symbols contained in the code (Tashtoush et al., 2013). Other factors also can affect the readability of the code. They are screen size, text font, text color, background color, use of spaces, etc. Each of us can have our own favourite screen size, text font and colour, but number of symbols is interpreted unambiguously. Apparently, the concise lambda expression has the following axiomatic properties:

- the lambda expression does not contain symbols[1] which do not impact the result of the lambda function;
- the lambda expression cannot be made shorter without changing its semantic meaning.

Syntax is the grammar, structure, or order of the elements in a language statement (Definition of syntax, 2017). Consequently, the perfect lambda syntax has the following properties:

[1] In special cases, tokens can be used instead of symbols.

- it allows to define any closed lambda expression[2] (an expression is closed if it has no free variables[3]);
- it allows to create concise lambda expressions.

Some might argue that token counting (in programming languages tokens are like words in natural languages) is better than symbol counting, because identifiers can be expressed in a more informative way. For example, identifier 'personX' is more descriptive than 'x'. Therefore, in this research two different forms of perfect lambda expressions are proposed:

- the perfect lambda expression containing minimal possible number of symbols;
- the perfect lambda expression containing minimal possible number of tokens.

Lambda expressions in the last few years have become a popular language construct and almost every major, mainstream programming language has introduced them. In some languages (JavaScript, Swift) lambda expressions are called "closures", in Kotlin, lambda expressions are called “function literals”. There exist many different names for the same semantic constructions. Therefore, the only way to think objectively about lambdas in different programming languages is to compare code snippets of different lambda syntaxes.

2. Lambda syntax evolution in C#

Lambda expressions in C# (C# 6.0, 2015, Meijer et.al., 2007) are very popular and a widely used feature. Data structures as shown in following code snippet are used in other code examples in this article to explain innovations of different lambda expression syntax:

```
using System;
class Person: IComparer<Person> {
    public string FirstName {get; private set;}
    public string LastName {get; private set;}

    public Person(string firstName, string lastName) {
        FirstName = firstName;
        LastName = lastName;
    }

    public int Compare(Person x, Person y) {
        return x.LastName.CompareTo(y);
    }
}
```

[2] Also called as closed lambda term.
[3] An instance of a variable is “free” in a lambda expression if it is not “bound” by a lambda. In following example: λy.xyz the variable x and z is free and y is bound by the lambda.

```
class Program {
    static void Main(string[] args) {
        var people = new List<Person>();
        //lambda expression code is supposed to be added here...
```

C# 2.0 introduced the concept of anonymous methods of writing unnamed inline statement blocks:

```
people.Sort(delegate(Person x, Person y) {
    return x.LastName.CompareTo(y.LastName);
});
```

C# 3.0 introduced lambda expressions, which are similar in concept to anonymous methods, but more expressive and concise:

```
people.Sort((Person x, Person y) => x.LastName.CompareTo(y.LastName));
```

The C# compiler can detect types of lambda expression parameters using type inference, so, in lambda expression declaration the parameter type information can be omitted, making lambda syntax even more concise and nice:

```
people.Sort((x, y) => x.LastName.CompareTo(y.LastName));
```

2.1. Possible improvements in C# lambda expressions

Implementation of the lambda expression syntax of C# contains parameter list declaration. The parameters are used in the body of the lambda. This means that the parameters list declaration repeats information present in the lambda body. In an ideal case, parameter declaration information should be inferred from its syntax of use. The theoretical solution is to get rid of the lambda parameters list and it can be done as follows:

```
people.Sort(=> x.LastName.CompareTo(y.LastName));
```

Unknown identifiers in the lambda body can be interpreted as lambda parameters, preserving the parameter order in lambda parameters list as they are used in the lambda body. Usage of meaningful names is better for code readability. Application of this principle to language specific keywords can increase readability in general, because mathematicians, programmers and users are familiar with that name. For example, lambda symbol "=>" can be renamed to a keyword "func" or "function" which is used to define functions in several programming languages, for example: Python®, JavaScript® and others. The improved code example is as follows:

```
people.Sort(func x.LastName.CompareTo(y.LastName));
```

The keyword "func" is needed to specify the beginning of the lambda expression, which leads to putting implicit parameters into the lambda expression parameters list instead of parameters list of the named method in which the code is written.

As C# lambda syntax can be made more concise, it is not perfect.

3. Java® 8: Introduction of "method references"

Java 8 lambda expressions (Gosling J. et.al. 2013) are like lambdas in C# 3.0. Following code snipped contains simple Java 8 code needed for explanation of further code examples:

```
import java.util.ArrayList;
import java.util.List;
import static java.util.Comparator.comparing;

public class Main {
  interface Person {
        String getFirstName();
        String getLastName();
    }
  public static void main(String[] args) {
    List<Person> people = new ArrayList<>();
    people.sort(comparing(p -> p.getLastName()));
  }
}
```

A construction named: "method references" is an interesting feature in Java 8 which allows to reference constructors or methods without executing them. The lambda expression shown in following example is simply a forwarder for the existing method 'getLastName'.

```
p -> p.getLastName()
```

Method references can be used to reuse the existing method instead of the lambda expression as shown in following example:

```
people.sort(comparing(Person::getLastName));
```

To implement their vision of the "method references" feature, language designers introduced the new operator "::". It signifies declaration of concise form of lambda expression, but the symbol "::" is placed in the middle of the expression. The expression starts with a desired type name, followed by the operator "::", and a method belonging to the type. Code readability may be a subjective matter, but humans mostly read from left to right.

Each "method reference" (lambda expression) can contain only one reference to some method, but in real situations often is necessary to combine calls of several methods in one lambda expression. Design of such a narrow feature raises language complexity, while improving code conciseness only in very specific use cases. As “method references” can’t express common real-world scenarios, “method references” does not fit the perfect lambda syntax definition.

3.1. Possible improvements in Java lambda expressions

Better syntactic form is demonstrated in following example:

```
people.sort(comparing(func Person.getLastName()));
```

The keyword "func" signifies the beginning of a concise form of lambda expression as described in section 2.1. The only unknown syntactic form so far is expression 'Person.getLastName', which is an instance method, but used as if it were a static method. Here type 'Person' is used to create auto-generated lambda expression parameter (also called anonymous implicit parameter (Vanags et.al., 2013)). Having a parameter of type “Person”, it is possible to access the instance method 'getLastName' as shown in following lambda expression:

```
func Person.getLastName();
```

It is equivalent to following method declaration:

```
void AutoGeneratedMethodName(Person autoGeneratedParameterName) {
    return autoGeneratedParameterName.getLastName();
}
```

The syntax improvement logic does not change in expressions consisting of several smaller lambda expressions as shown in following example:

```
people.sort(comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

If there exist two different lambdas, then the keyword "func" should be used twice as demonstrated in following example:

```
people.sort(comparing(func Person.getLastName)
    .thenComparing(func Person.getFirstName));
```

More importantly, "method references" can't be combined to create one lambda expression reusing more than one method, as shown in following example:

```
people.sort(func Person.LastName.CompareTo(Person.LastName));
```

Instead of using a comparator, complex lambda expression is used to compare two different person instances (instance comparison logic is assumed to be like the examples demonstrated in section 2. The main purpose here is to demonstrate lambda syntax improvement rather than discussing better functional approach in designing comparators).

In previous example, the first usage of type Person is supposed to generate the first parameter for lambda expression. The second usage of type Person is supposed to generate the second parameter for the same lambda expression. Equivalent code demonstrating lambda expression syntax using explicit parameters is shown in following example:

```
people.sort((Person a, Person b) ->
a.getLastName().CompareTo(b.getLastName()));
```

4. Kotlin®: Introduction of operator "it"

Kotlin (Kotlin, 2015; Jemerov and Isakova, 2017) is a statically-typed programming language that runs on the Java Virtual Machine. Higher order functions in Kotlin can be declared as shown in following example:

```
//accepts parameter which should be function accepting 0 parameters
fun test0(myFunc: () -> Unit) {}
//accepts parameter which should be function accepting 1 parameter
fun test1(myFunc: (Int) -> Unit) {}
//accepts parameter which should be function accepting 2 parameters
fun test2(myFunc: (Int, Int) -> Unit) {}
```

Kotlin supports abstraction called "function literal" which is an "anonymous function" or "lambda expression", i.e. a function that is not declared, but passed immediately as an expression. A function literal is always surrounded by curly braces. Parameter declarations in the full syntactic form go inside parentheses and have optional type annotations. The body goes after a "->" sign. Function literals can be used to call higher order functions as shown in following example:

```
test0 { }
test1 { parameterName -> parameterName + 2 }
test2 { it, it2 -> it + it2 }
```

It's very common that a function literal has only one parameter. If Kotlin can figure the method signature by itself then it allows programmers not to declare the only parameter, and will implicitly declare it for programmers under the name "it". Full form using explicit parameter declaration is shown in following example:

```
test1 { it -> it + 2 }
```

Example using keyword "it" to access the only parameter of function literal is demonstrated in following example:

```
test1 { it + 2 }
```

Function literal can be passed to a single parameter function without accessing the parameter as shown in following example:

```
test1 { }; //compiles successfully
```

This leads to conclusion, that keyword "it" does not affect creation of function parameters list. Keyword "it" is used to access the only parameter of "function literal". It is also not supported in case of functions with accepts more than one or less than one parameter. Examples of invalid usage of keyword "it" with resulting error messages are the following:

```
test0 { it + 2 } //Unresolved reference: it
test1 { parameterName + 2 } //Unresolved reference: parameterName
test2 { }; //Expected 2 parameters of types Int, Int
test2 { it + it2 } //Expected 2 parameters (Int, Int); Unresolved
references: it, it2
```

Syntax of keyword "it" looks like a perfect solution, but it is restricted to single parameter functions, which is not enough abstract solution. Operator "it" is a significant improvement over Oracle's vision "method references", because keyword "it" can be reused more than once in the same function literal. Operator “it” is limited to single parameter function literals therefore it does not fit the perfect lambda syntax definition.

5. Q: concise syntax support for up to 3 implicit parameters

In programming language Q (Tutorials/Functions, 2016), function “f” accepting three parameters can be defined and executed as follows:

```
f:{[a;b;c]a+b+c}
f 1 2 3
```

Parameters “a”, “b” and “c” are declared explicitly using square brackets. Programming language Q can recognize up to eight explicitly declared parameter names and up to three implicit parameters (x, y and z). Implicit parameter “x” is special symbol to access the first parameter of the function, implicit parameter “y” is special symbol to access the second parameter of the function and “z” is special symbol to access the third parameter of the function. Therefore, previously defined function “f” can be rewritten using implicit parameters:

```
f:{x+y+z}
f 1 2 3
```

If in real world 3 would be maximum number of function parameters, then Q implicit parameters could be used to create lambda syntax which fits the perfect lambda syntax definition.

6. Swift®: Introduction of shorthand argument names

Closures in Swift (Swift 3.0, 2017) are like lambdas in other programming languages. Swift's standard library provides a function called 'sorted', which sorts an array of values of a known type based on the output of provided sorting closure. The following example demonstrates a closure expression passed as argument to function 'sorted':

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
var reversed = sorted(names, { (s1: String, s2: String) -> Bool in
return s1 > s2 } )
```

Because the sorting closure is passed as an argument to a function, Swift can infer the types of its parameters and the type of the value it returns from the type of the 'sorted' function's second parameter. Because all the types can be inferred, the return arrow "->" and the parentheses around the names of the parameters can also be omitted as shown in following example:

```
var reversed = sorted(names, { s1, s2 in return s1 > s2 } )
```

Swift automatically provides "shorthand argument names" to inline closures, which can be used to refer to the values of the closure's arguments by the names $0, $1, $2, and so on.

If shorthand argument names are used within closure expression, closure's argument list can be omitted, and the number and type of the "shorthand argument names" will be inferred from the expected function type. Keywords 'in' and 'return' can also be omitted, because the closure expression is made up entirely of its body (as demonstrated in following example).

```
var reversed = sorted(names, { $0 > $1 } )
```

Here, $0 and $1 refers to the closure's first and second String arguments.

So far, it is the best lambda expression syntax reviewed (except improvements sections).

Imagine universal operator "it" for accessing any anonymous function parameter. This can be achieved by adding parameter index to keyword "it". The result would be it0, it1, it2, it3 and so on… Now replace keyword "it" with symbol "$" and the result will be Apple's improvement over JetBrain's vision about the perfect lambda syntax. The syntax is concise, it contains no redundant parts, but code readability suffers, because without knowing the context, it is difficult to understand the meaning of parameters it0 and it1 or $0 and $1.

6.1. Possible improvements in Swift lambda expressions

For more than 400 years, mathematicians have used symbols instead of indexes to denote unknown quantities (Stansifer, 1994). Code readability (Tashtoush et.al., 2013) could be improved, if it would be possible to provide names for lambda expression parameters instead of indexes as shown in following example:

```
var reversed = sorted(names, { current > next } )
```

Meaningful parameter names allow us to understand purpose of the parameters without digging deep into source code or accompanied documentation. Meaningful construction names can reduce the amount of needed code comments and documentation.

Besides, if symbol $ is used to define other constructions too, then expression '$0' consists of two tokens: a special symbol $ and a number. While construction 'x' represents only one token - an identifier. If 'x' is not reserved symbol in the programming language, then even expression 'x23' is considered as one token - an identifier. Swift uses symbol $ only for defining implicit parameter names and expression $number can be processed as one token, therefore Swift lambdas are tokens perfect, but not symbols perfect, because Swift implicit parameter names consist of at least two symbols and can be made shorter.

It is possible to use language features with low readability factor, but it's impossible to use such features if they are not enough expressive. In case of usage of nested lambda expression, it is impossible to unambiguously define nested anonymous function shown in following example, using only shorthand argument names.

```
var outerFunc: (Int) -> Int =
{
    (a:Int) in
    var z = a
    var innerFunc: (Int) -> Int = { (b:Int) in return a + b}
    return z + innerFunc(5)
}
```

The best thing that can be achieved: use of shorthand argument names in inner (nested) lambda expression and avoiding the use of shorthand argument names in outer function as shown in following example:

```
var outerFunc: (Int) -> Int =
{
    (a:Int) in
    var z = a
    var innerFunc: (Int) -> Int = { a + $0 }
    return z + innerFunc(5)
}
```

Because shorthand argument names always point to closest lambda expression scope, there is no way to instruct compiler that construction "$0" in inner function should result in accessing parameter of outer function parameters list. The problem can be solved by:

- using parameter names instead of parameter indexes;
- and for each function picking different parameter names as shown in following example:

```
var outerFunc: (Int) -> Int =
{
    var z = a;
    var innerFunc: (Int) -> Int = { a + b }
    return z + innerFunc(5)
}
```

In previous example, each unknown identifier is interpreted as parameter of closest lambda expression. There is one unknown identifier in 'outerFunc' scope which is identifier 'a' and one unknown identifier in 'innerFunc' scope where unknown identifier name is 'b'. That means, that 'outerFunc' accepts one parameter named 'a' and 'innerFunc' accepts one parameter named 'b'.

7. Lambda expression syntax innovations in functional programming languages

Functional programming languages contain several interesting features, which can be used in improving the lambda syntax.

7.1. Scala implicit parameters

The Scala language specification (Scala, 2017) supports the feature called "implicit parameters", which will be called in this paper as "Scala implicit parameters". It is called as such because Scala implicit parameters are declared outside the method scope. Historically, the idea of parameters is related to the method. If parameters are declared outside of a method, then they are not parameters of the method and they are variables of a scope higher than the scope of the method (usually a higher scope is a global scope and parameters defined in a global scope are called global variables). Scala implicit parameters allow the declaration of the last parameter list of a function to be implicit. The syntax of Scala implicit parameters is demonstrated in the following example (using Scala REPL):

```
scala>   def   speakImplicitly   (implicit   greeting   :   String)   =
println(greeting)
speakImplicitly: (implicit greeting: String)Unit

scala> speakImplicitly("Goodbye world")
Goodbye world

scala> speakImplicitly
error: could not find implicit value for parameter greeting: String

scala> implicit val hello = "Hello world"
hello: String = Hello world

scala> speakImplicitly
Hello world
```

It is possible to call function 'speakImplicitly' as normal but, it is also possible to leave out the implicit argument list. The Scala compiler will look for a value in the enclosing scope, which has been marked as implicit. If we try to do that and there is no such value in the scope, then the compiler will return error code.

Scala implicit parameters are type safe and they are selected based on the static type of the arguments. This means that Scala implicit parameters allow reuse implicit parameter in the different functions. However, if there are multiple Scala implicit parameters of the same type, compiler will fail to use any, because it is not able to choose between them as demonstrated in following example:

```
scala>   def   speakImplicitly   (implicit   greeting   :   String)   =
println(greeting)
speakImplicitly: (implicit greeting: String)Unit

scala> implicit val param1 = "hello";
param1: String = hello

scala> implicit val param2 = "world";
param2: String = world

scala> speakImplicitly
error: ambiguous implicit values:
    both value param1 of type => String
    and value param2 of type => String
    match epected type String
```

In Scala, implicit parameter idea use cases are significantly limited and it is similar to global variables, which in general is considered as bad practice (Parkhe, 2013). The Scala implicit parameters idea can be considered as a leaky abstraction (Spolsky, 2002), Therefore, it is worth to try to look for a better implicit parameter abstraction.

7.2. Clojure shorthand lambda syntax

In programming language Clojure (Tayon and Fingerhut, 2016) function literal can be defined using syntax as demonstrated in following example:

```
#(...)
```

Inside lambda body construction %N can be used, which defines the value of anonymous function argument N. Symbol % is short form for %1 - the first argument. Example of lambda, which accepts three arguments and performs summing operation, looks like this:

```
# (+ % %2 %3)
```

Apparently, the idea of Clojure shorthand lambda syntax is the same as Swift's closure syntax using shorthand argument names. As Clojure is older programming language than Swift, it seems that Swift creators have been influenced by Clojure syntax.

8. Combining the idea for a perfect lambda syntax acquisition

Lambda expression improvements using implicit parameter full form (Vanags et.al., 2013) and keyword 'func' to indicate beginning of lambda expression are demonstrated in following example.

```
func (Person x).LastName.CompareTo((Person y).LastName)
```

Using type inference the compiler can detect parameter types from the method 'Sort' signature. Therefore, previous example can be improved by omitting parameter type declaration and using implicit parameters in canonical form as shown in following example:

```
func x.LastName.CompareTo(y.LastName)
```

As parameters 'x' and 'y' are not reused in the body of the lambda, they can be declared in anonymous implicit parameter form as follows:

```
func Person.LastName.CompareTo(Person.LastName)
```

If it is desired to sort list of people in opposite direction, then lambda parameter order should be changed and it can be done using Grace~ operator (Vanags, 2015) postfix form as shown in following example:

```
people.Sort(func Person~.LastName.CompareTo(Person.LastName));
```

or using Grace~ operator prefix form:

```
people.Sort(func Person.LastName.CompareTo(~Person.LastName));
```

If lambda body consists of more than one expression-statement, code block can be used in lambda declaration as it is shown in following example:

```
people.Sort(func {
        return Person.LastName.CompareTo(Person.LastName);
    });
```

Compare example of 'method references':

```
people.sort(comparing(Person::getLastName));
```

with example demonstrating usage of 'implicit parameters':

```
people.sort(comparing(func Person.getLastName()));
```

It seems that "method references" syntax is focused on syntax conciseness, but method references work only on single methods and their use cases are limited. "Implicit parameters" does not have such limitation.

Swift programming language has significant language design restriction to use shorthand argument names in nested lambda expressions. To overcome that restriction, programmers needs to do some hacks. Following example demonstrates nested function with use of "shorthand argument name" only in inner anonymous function.

```
var outerFunc2: (Int) -> Int =
{
    var outer = $0
    var innerFunc: (Int) -> Int = { outer + $0 }
    return innerFunc(5)
}
```

Implicit parameters in opposite to "shorthand argument names" can be freely used in both scopes: outer and inner anonymous function scope. Using implicit parameters together with parameter scope correction operator Grace~, code can be made even more concise and more readable as demonstrated in following example.

```
var outerFunc2: (Int) -> Int =
{
    var innerFunc: (Int) -> Int = { ~outer + inner }
    return innerFunc(5)
}
```

Implicit parameters and Grace~ operator are needed features to achieve the perfect lambda syntax. The perfect lambda syntax can't contain explicit declaration of function parameters list. Therefore, in perfect lambda implicit parameters will be used. Grace~ operator consumes less number of symbols or tokens to change parameters order than usage of explicit lambda parameters list declaration.

Implicit parameters still have one design restriction, it is their inability to define methods which accept a parameter and do not use that parameter in the method body. Example method shown in following code fragment can't be defined using only implicit parameters:

```
function PrintFirstName(string firstName, string lastName) {
    Write(firstName);
}
```

Defining unused parameters explicitly is not compatible with the definition of the concise lambda and perfect lambda syntax. The perfect lambda syntax should not encourage users to write unused parameters explicitly. Writing unused parameters implicitly means assuming, that each lambda expression has infinite number of unused parameters. The real maximal number of possible unused parameters depends on the lambda usage context. If the lambda usage context, for example - function type to which the lambda will be assigned accepts 1 parameter, then the maximum possible number of unused parameters is 1, see following C# example with anonymous delegate (predecessor of the C# lambda expressions):

```
   Action<int> func2 = delegate { };
```

Theoretically, it is possible to improve C# syntax to allow implicit unused parameters as demonstrated in following example:

```
Action<int> func1 = {  };
```

Therefore, the function PrintFirstName can be expressed in lambda using implicit parameters as follows:

```
Action<string, string> printPartOfName = Write(firstName);
```

The problem still exists for untyped programming languages, because you can’t get type information from the usage context (how many parameters, what are parameter types) using the type inference. And how to define the lambda to match the method PrintLastName (shown in the next example) printing only the parameter “lastName”?

```
function PrintLastName(string firstName, string lastName) {
    Write(lastName);
}
```

The problem can be solved by using parameter ignorance operator # - the idea is similar to parameter order correction operator # (Vanags, 2015). However, parameter ignorance operator is used to ignore parameters instead of affecting the parameter order.

Parameter ignorance operator # should be used as prefix or postfix part of any valid expression or subexpression. Its expression postfix syntax is following:

```
expression#N
```

and expression prefix syntax is following:

```
N#expression
```

where N is optional part and it determines the number of repeated usages of parameter ignorance operator #.

Function PrintLastName can be replaced with equivalent lambda using parameter ignorance operator as follows:

```
Action<string, string> PrintFirstName = Write(firstName)#;
```

or using parameter ignorance operator in subexpression as follows:

```
Action<string, string> PrintFirstName = Write(firstName#);
```

Delegate PrintLastName (equivalent to function PrintLastName) can be initialized with lambda expression as follows:

```
Action<string, string> PrintLastName = #Write(lastName);
```

To ignore several parameters with one usage of symbol #, after the parameter ignorance operator # can be declared number of parameters to which the parameter ignorance operator is applied.

Few unanswered questions are: "Does the perfect lambda contain lambda symbol? Maybe, the perfect lambda syntax can consist only of function body declaration (which of course should be perfect too)?" Swift's "shorthand argument names" usage demonstrates that lambda symbol can be inferred from the lambda usage context. Therefore, the lambda expression, which contains lambda symbol, is not perfect by the perfect lambda definition. General-purpose programming languages may need lambda symbol to be able to distinguish lambdas from other abstractions.

8.1. Minimal set of features for the perfect lambda syntax

So far, we have explored implicit parameters, Grace~ operator and parameter ignorance operator #. This chapter answers the questions: Do we need to use all these features to get the perfect lambda syntax, and are these features enough to express any computable function?

The simplest function to test is identity function (or I combinator), which in pseudocode can be expressed as follows:

```
func (x) { return x; }
```

Identity function in lambda calculus is declared as follows:

```
λx.x
```

The shortest possible definition of identity function is following (using implicit parameters abstraction):

```
x
```

Apparently, there are nothing more to remove without losing the semantics of identity function, therefore, it is an example of symbols perfect lambda expression. And it means, that implicit parameters can make the lambda syntax perfect.

The second simplest function to test is function which represents K combinator. In pseudocode, it can be expressed as follows:

```
func (x, y) { return x; }
```

K combinator function in lambda calculus is declared as follows:

```
λx y.x
```

If we have type system, and the lambda usage context can tell us that the function accepts 2 parameters, then the shortest form to define K combinator in lambda expression using implicit parameter is following:

```
x
```

Without the information of the lambda usage context, previous example is equivalent to the identity function causing ambiguity. Therefore, at least one more symbol is needed to get rid of ambiguity in the shortest K declaration. The solution is usage of parameter ignorance operator #, see following example:

```
x#
```

Therefore, the parameter ignorance operator # is part of the perfect lambda syntax. In typed programming languages, sometimes it is possible to infer unused parameters from the function usage context. If the programmer can reorder parameters or decompose bigger function (accepting more parameters) into smaller functions (accepting less parameters) then the operator # usage can be avoided, but parameter reordering might not always be possible and it might require more symbols and tokens to define decomposed functions.

The next function to test is the function which represents S combinator. In pseudocode, it can be expressed as follows:

```
func (f, g, x) { return f(x)(g(x)); }
```

S combinator in lambda calculus is declared as follows:

```
λf g x.f x(g x)
```

Using implicit parameters, the short declaration of S combinator function in lambda calculus could be the body of the S combinator lambda expression, but we need to fix the parameter order. Therefore, at least one more symbol is needed to declare S combinator lambda according to the perfect lambda syntax and the following example demonstrates how that can be achieved using implicit parameters and Grace~ operator:

```
f x(~g x)
```

In some cases, the programmer can reorder parameters and avoid using Grace~ operator, but it is not always that the programmer can modify existing contracts. Therefore, Grace~ operator is part of the perfect lambda syntax.

The only syntactic construction that has left unmodified in lambda calculus syntax is bracket usage to change the operation priority. Two symbols for brackets abstraction in S combinator lambda expression is long. In such simple expression, the operation priority change could be expressed with one symbol. Therefore, lambda expressions containing brackets are not perfect. Bracket syntax perfection is part of the future research.

In this chapter, we demonstrated that combinators K and I can be expressed in the shortest possible lambda expressions using implicit parameters and parameter ignorance operator. Combinator S can be made shorter using implicit parameters and Grace~ operator.

As the perfect lambda syntax does not contain lambda symbol, we need separator symbol to be able to declare several lambdas in one expression (use of lambda calculus operation – application). Such separator symbol can be comma – ‘,’.

By using SKI or SK combinator base, it is possible to express any closed lambda expression (Hankin, 2004). Therefore, implicit parameters, Grace~ operator, parameter ignorance operator and lambda expression separator symbol can be used to define any closed lambda expression. It does not mean, that all the lambdas declared using implicit parameters, Grace~ operator or parameter ignorance operator will be the shortest, but using these constructions it is possible to declare short lambda expressions and often they are the shortest.

9. Summary

Lambda syntax in programming languages can be classified in four different categories:

- Lambda syntax consisting of explicit function parameters list and function body (C#, Java);
- Lambda syntax consisting of function body from within function parameters are accessed using special reserved keywords. The number of needed special keywords depends on the number of parameters supported in programming language lambda expressions. (Kotlin has support for only 1 parameter concise lambdas using keyword ‘it’ for the first parameter. Programming language Q has support for up to 3 parameters concise lambdas using keywords: ‘x’ – first parameter, ‘y’ – second parameter, ‘z’ – third parameter (programming language Q);

- Lambda syntax consisting of function body from within function parameters are accessed using special symbol and index (Clojure, Swift);
- Lambda syntax consisting of function body, and each unknown identifier is interpreted as function implicit parameter. Parameter order depends on the order in which parameters are used in the function body and the parameter order can be corrected using Grace~ operator. (language: Kat-lang, implementation: Il Calculus) (Il Calculus 2015).

Implicit parameters together with Grace~ operator and parameter ignorance operator can be used to improve lambda expression syntax in programming languages. Often, the usage of these constructions makes it possible to achieve the perfect lambda syntax – the syntax allowing to write any closed lambda expression without containing redundant syntactic parts.

The number of symbols is a fundamental aspect of code readability (Tashtoush et al., 2013). In general, the shorter and more compact the code, the higher the code readability factor. Therefore, the perfect lambda syntax makes lambda expressions more readable and improves code editability factor (Blow, 2014) allowing programmers to be more productive in their work.

Programming languages can support many different abstractions and sometimes syntax may not satisfy perfect conditions for all the supported abstractions. It might be that some programming language syntax can't be improved to support the perfect lambda syntax without limiting support of some other abstractions.

Acknowledgements

This research started as a part of Mikus Vanags' PhD thesis "Abstract data processing technologies" available at: https://doi.org/10.22616/lluany/2017.010

We want to thank to all the reviewers of our paper. You helped us a lot!

References

Atencio, L. (2015). Understanding Lambda Expressions. Retrieved from: https://medium.com/@luijar/understanding-lambda-expressions-4fb7ed216bc5#.liszqjahd

Blow, J. (2014). A programming language for games, talk #2

C# 6.0 (2015). C# Language Specification, Version 6. Microsoft Corporation. Retrieved from https://github.com/ljw1004/csharpspec/blob/gh-pages/README.md

Definition of syntax. 2017. WhatIs. Retrieved from: http://whatis.techtarget.com/definition/syntax

Dushkin, R.V. (2006). Functional progamming in Haskell (in Russian: Душкин Р.В., Функциональное программирование на языке Haskell. ДМК Пресс).

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. (2013). The Java® Language Specification, Java SE 8 Edition; Oracle America, Inc. Retrieved from https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Hankin, C. (2004). An Introduction to Lambda Calculi for Computer Scientists. College Publications.

IlCalculus. 2015. Logics Research Centre. Retrieved from https://www.microsoft.com/en-us/store/p/il-calculus/9wzdncrdmb09

Jemerov, D, Isakova, S. (2017). Kotlin in Action. Manning Publications.

Kotlin language reference (2015). JetBrains. Retrieved from: https://kotlinlang.org/docs/kotlin-docs.pdf

Martin, R.C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. p. 95. ISBN 978-0135974445.
Meijer, H., Hejlsberg, A., Box, D., Warren, M., Bolognese, L., Katzenberger, G., Hallam, P., Kulkarni, D. (2007). Lambda expressions. US Patent App. 11/193,565.
Parkhe, R. (2013). Global Variables Are Bad. Retrieved from http://wiki.c2.com/?GlobalVariablesAreBad
Rojas, R. (1998). A Tutorial Introduction to the Lambda Calculus. Berlin. Retrieved from http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf
Scala language documentation. (2017). Retrieved from http://www.scala-lang.org/documentation/
Spolsky, J. 2002. The Law of Leaky Abstractions. Retrieved from https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/
Stansifer, R. D. (1994). The study of programming languages. Prentice-Hall, Inc. New Jersey, USA.
Swift 3.0 (2017). The swift programming language. Apple Inc. Retrieved from: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html
Tashtoush, Y. Odat, Z., Alsmadi, I., Yatim, M. (2013). Impact of programming features on code readability. International Journal of Software Engineering and Its Applications, 7(6):441 458, 2013.
Tayon, S., Fingerhut, A. (2016). Clojure 1.8 Cheatsheet v35. Retrieved from: https://clojure.org/api/cheatsheet
Tutorials/Functions (2016). Kx. Retrieved from: http://code.kx.com/wiki/Tutorials/Functions
Vanags, M. (2015). Grace operator for changing order and scope of implicit parameters, August 16 2016. US Patent 9,417,850.
Vanags, M., Justs, J., Romanovskis, D. (2013). Implicit parameters and implicit arguments in programming languages. June 7 2016. US Patent 9,361,071.
Veitch, J. (1998). "A history and description of CLOS". In Salus, P. H. Handbook of programming languages. Volume IV, Functional and logic programming languages (First Ed.). Indianapolis, IN: Macmillan Technical Publishing. pp. 107–158. ISBN 1-57870-011-6.

Received October 18, 2017, revised February 27, 2018, accepted March 2, 2018