ECOTE Project (A21): Program that converts C# code with lambda expressions into code without lambda expressions

Author: Paweł Paczuski
Supervisor: dr inż. Anna Derezińska

May 6, 2017

Contents

1 Theoretical introduction **2**
1.1 Lambda calculus in Mathematics . . . 2
1.2 Lambda calculus in programming . . . 2
1.3 Lambda expressions in C# . . . 2
1.3.1 Simple lambda expressions . . . 3
1.3.2 Parenthesized lambda expressions . . . 3
1.3.3 Statement lambdas . . . 3
1.3.4 Lambda expressions capturing local variables . . . 3
1.3.5 Recursive lambda expressions . . . 3

2 Lambda Converter for C# **4**
2.1 Idea . . . 4
2.2 Transformation stages . . . 4
2.2.1 Syntactic analysis . . . 4
2.2.2 Semantic analysis . . . 4
2.2.3 Code transformation . . . 4
2.3 Transformation algorithm . . . 4
2.4 Roslyn . . . 5
2.4.1 Motivation . . . 5
2.4.2 Architecture . . . 5
2.5 Limitations . . . 6
2.6 Errors . . . 6
2.7 Tests . . . 6
2.8 Performance considerations . . . 6
2.9 Practical applications . . . 7
2.10 Exemplary input and output . . . 7
2.10.1 Input . . . 7
2.10.2 Output . . . 9
2.11 Lambda Transformer code . . . 14
2.11.1 Program.cs . . . 14
2.11.2 LambdaConverter.cs . . . 15
2.11.3 TransformationInfo.cs . . . 20

1

Theoretical introduction

1.1 Lambda calculus in Mathematics

Lambda calculus was introduced as a mean of formalizing the concept of expressing computation based on function abstraction and application using variable binding and substitution long before computers and programming languages existed. [1] The central concept in λ calculus is "expression". A "name", also called a "variable", is an identifier which, for our purposes, can be any of the letters a,b,c,... An expression is defined recursively as follows[2]:
<expression>:=<name> | <function> | <application>
<function >:= λ <name> . <expression>
<application>:=<expression><expression>

The evaluation of lambda
λ calculus can be used to express any computable function. Because of this fact it is equivalent to Turing machines but the computational power of λ calculus is not connected in any way with hardware implementation. This concept is more likely to be implemented using software.

1.2 Lambda calculus in programming

In spite of being object-oriented, languages C#, JAVA and C++ borrow several concepts from functional paradigm. One of them is the concept of anonymous function. Exemplary situation in which anonymous functions are useful can be filtering array using some set of predicates. Existing array is being iterated over and a function is called with each element of the array as its argument. The function examines properties of the element and returns boolean value. If the function returns true, the element is being pushed to the array that is result of filtering the former one. Otherwise, this element is skipped. Using purely object-oriented approach, one would have to create an implementation of an interface that has a function that returns boolean value and is able to decide whether an element of the array satisfies all conditions. The problem with this solution is the distance between code that makes decision about an element and the place where function that filters the array was called. When somebody reads code written in this manner, the have to go to the definition of class and take a look at implementation of the interface method. A lot of boilerplate code is introduced and namespace is polluted with simple classes that provide no special functionality. This is the place where traditional λ calculus gives practical tool that increases language expressiveness and therefore allows for writing code that is easier to read and has more bugs. One could define anonymous function that decides about filtering an item in the place of calling filter function making the code much more expressive. This idea was implemented in C# language mechanism called LINQ

1.3 Lambda expressions in C#

Creators of C# language decided to split syntax of a lambda expressions into several types. Lambda expressions in C# language consist of arguments, => operator and expression or statement block. The => operator has the same precedence as assignment (=) and is right associative [4]

1.3.1 Simple lambda expressions

The simplest lambda expressions. They are constituted of a single argument, => operator and expression. Types of arguments and returned value can be inferred.

Listing 1.1: This lambda has single integer argument m and returns an integer m - 1

```
Func<int, int> lambd1 = m => m - 1;
```

1.3.2 Parenthesized lambda expressions

When lambda has more than one argument, parentheses have to be present. Single-argument lambda expression with parentheses also belongs to this category.

Listing 1.2: Lambda arguments can have types defined explicitly

```
Func<int, int, int> lambd2 = (int m, int n) => (m - n);
```

1.3.3 Statement lambdas

More complex lambdas can contain statements within their body so, in addition to return value, they can have side effects.

Listing 1.3: execution lambd3 has side-effects that have no impact on the returned value

```
string text = "abc";
Func<int, int> lambd3 = (n) =>
{
    Console.WriteLine(text);
    Console.WriteLine(n);
    return n % 2;
};
```

1.3.4 Lambda expressions capturing local variables

Lambda expressions can implicitly capture local/instance variables. This has some serious implications e.g. when lambda expression capturing variable in a local block is assigned to a field of a class and then called in a different context, it has to be able to read value of a variable that used to be present in scope where it was assigned to the class field.

Listing 1.4: "lambd4 has access to the local variable text in any context that. This property is sometimes called closure"

```
    string text = "this is some local text";
    Func<string, string> lambd4 = (n) => (text+n);
```

1.3.5 Recursive lambda expressions

These are lambda expressions that can define other lambda expressions in their body. Due to limited practical applications (their syntax is not easy to read which is contrary to the expectation that lambdas are used to make code more readable), they are not described in a detailed way in this document.

Listing 1.5: "factorial defined using lambda"

```
    Func<Func<int, int>, Func<int, int>> factorial = (fac) => x => x == 0 ? 0 : x * fac
        (x - 1);
```

2

Lambda Converter for C#

2.1 Idea

The goal is to create a program (from now on referred to as Converter) that as an input receives source code written in C# language that may contain lambda expressions of various kind. The program performs transformations of several kinds on the code in order to output modified version (but having the same functionality) of the code that contains no lambda expressions.

2.2 Transformation stages

The inner workings of the program can be split conceptually into several stages:

1. Syntactic analysis
2. Semantic analysis
3. Code transformation

2.2.1 Syntactic analysis

In this stage the source code is analyzed in order to create syntax tree of the code. This data structure keeps all information about syntactic constructs in the input code. For a given syntax tree node, one can get information about its type, access modifier, identifier etc.

2.2.2 Semantic analysis

Lambda conversion may require type inference for arguments and results, data flow information (what local scope variables are used in lambda's body), etc. These types of information can be taken from semantic model that is build using previously obtained syntax tree.

2.2.3 Code transformation

Information gathered in previous steps is used to find all lambda expressions in code that the program is capable of transforming. For each node transformation algorithm (described in section 2.3) is applied. After all compatible lambda expressions have been replaced, the resulting syntax tree is being converted into code it represents, and resulting code is reformatted in order to increase readability.

2.3 Transformation algorithm

1. Determine which kind of lambda expression syntax described in section 1.3 current lambda has
2. Determine names and types of lambda expression arguments

3. Analyze what variables from the scope are referenced in lambda's body
4. Create syntax node for a class declaration, give it unique identifier
5. Add fields to the class syntax node that correspond to types of variables and names of variables referenced in lambda's body
6. Create method syntax node that has the same return value as lambda, receives the same arguments and infer types of lambda's arguments. Its identifier can be e.g. "LambdaMethod". For statement lambda, copy original body into new method. For expression lambdas, add return keyword and concatenate with original body.
7. Insert method syntax node into newly created class syntax node
8. Insert class declaration syntax node into body of a class that contains lambda expression
9. Find last expression or statement before original lambda expression.
10. Create syntax nodes that correspond to instantiation of class, filling fields with values of variables from current scope. Insert those nodes just after the previously found node.
11. Create method reference syntax node. Replace syntax node that used to define lambda expression with method reference node.

2.4 Roslyn

2.4.1 Motivation

The problem of converting code containing lambda expressions has a lot of inherited complexity. The C# language is a multi-paradigm language that evolved since its first release. Therefore, a grammar that defines it has many productions. The task of writing lexer and parser on our own as a subtask of the whole project would take a lot of time and effort to finish. Luckily, the creators of C# language provide a set of open-source tools used to analyze and transform code called Roslyn. Roslyn exposes the C# compiler's code analysis to a programmer as a consumer by providing an API layer that mirrors a traditional compiler pipeline. [5] Roslyn is used in the first two stages of operation of the Converter described in 2.2.

2.4.2 Architecture

Roslyn consists of two layers of API: compilers API and workspaces API.
Compilers API provides programmer with object models that are used at each stage of the compilation: syntactic and semantic.
Workspace API provides an object model to aggregate the code model across projects in a solution. Mainly for code analysis and refactoring around IDEs like Visual Studio, though the APIs are not dependent on Visual Studio[7].

Syntactic model

Syntactic model is represented mainly by a syntax tree whose nodes are called syntax nodes-constructs that represent syntactic elements such as statements, declarations, clauses, expressions, etc.
Implementation of syntax tree used in Roslyn is immutable, which means that any change to nodes of the tree results in creation of a clone of the tree with selected nodes different. This allows for parallelism as most of Roslyn APIs are thread-safe.

Semantic model

Semantic model adds to the syntactic model about language rules and how the information flows in regions of source code. Semantic model uses symbols as representations of a distinct element declared by the source code or imported from an assembly as metadata. Every name-space, type, method, property, field, event, parameter, or local variable is represented by a symbol[6].

Workspace

Workspace is an abstraction that describes solution as a collection of projects that contain documents with source code. This API is used in Converter in order to make syntax node insertion easier and efficient.

2.5 Limitations

In spite of having the capability of capturing local variables to be used in lambda body, Converter lacks the functionality that would capture fields of a class in which lambda expression is defined.
Some minor differences can be observed when one deals with C# value types (float, double). They are internally implemented as structs named Single, Double respectively. This has no direct effects on the execution of output code.

2.6 Errors

Main class of errors that can occur while executing Converter are those caused by improper syntax of the code.

2.7 Tests

This section contains description of scenarios that were taken into consideration while developing the program. Code that implements these cases is included in section 2.10.1 and results of conversion are included in section 2.10.2. In order to increase correctness of the final solution, tests were written before the actual program.

- Test if SimpleLambdaExpressions are converted properly
- Test if ParenthesisedLambdaExpressions are converted properly
- Test if lambda expression containing statements are converted properly
- Test if lambda expressions inserted into LINQ constructs are converted properly
- Test if arguments of the lambda expressions are inferred
- Test if arguments with types specified are used properly
- Test if Converter captures all variables that are used in lambda body
- Test if Converter signals an error when lambda captures field of the ancestor-class
- Test if Converter signals an error when input code is not a proper C# code
- Test if Converter deals with conversion of reference types, value types, built-in types, user-defined types
- Test if Converter instantiates generated classes properly, all fields are filled
- Test if Converter replaces lambda expression with method
- Test if Converter reformats the output code
- Test if all fields of generated class are public

2.8 Performance considerations

Due to immutability of the syntax tree implementation provided by Roslyn, each insertion creates a new snapshot of the tree. This has to be taken into consideration when inserting/removing nodes. Solution to it is to use built-in mechanism of batch-insertions. Because each modification creates new tree, a remembered place in the code is lost. To keep track of changes DocumentEditor abstraction can be used. It is helpful when one deals with many modifications but requires usage of Workspace API that has performance drawbacks.

2.9 Practical applications

Increasing popularity of refactoring tools like JetBrain's Resharper or tools included in Visual Studio installation like Intellisense or refactor-suggestions make it quite possible that Lambda Converter could be used in this kind of tooling to allow for quick transformations of portions of code. Program in its current form would be quite easy to convert into a Visual Studio extension.
One could also use Lambda Converter to teach about lambda expressions and how they can be implemented in programming languages.

2.10 Exemplary input and output

2.10.1 Input

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace lambda_converter.target_code
{
    class LambdaCode
    {
        List<int> ints = new List<int> { 1, 356, 23, 1, 56, 2, 123, 555, 78, 221, 4, 0, 2,
            5, 1 };

        int someClassField = 100;
        int anotherClassField = 3;
        int modifiedClassField = 0;

        class UserDefinedType
        {
            public int a;
            float c;

        }

        public void Method()
        {

            //SimpleLambdaExpression
            var even = ints.Where(m => m % 2 == 0).ToList();

            int[] externalInts = { 1, 3, 5 };
            int[] localInts = { 3, 6, 9 };

            //ParenthesisedLambdaExpression
            var zipped = localInts.Zip(externalInts, (int m, int n) => { return m - n; }).
                ToList();

            string text = "result of zipping";
            string teee = "some random text";
            int abba = 123;
            //statement lamda with capture
            zipped.ForEach((n) =>
            {
                Console.WriteLine(text);
                Console.WriteLine(n);
                Console.WriteLine(teee + 3 * abba + text);
            });

            Func<int> voidLam = () => 3;

            //Check if custom reference types converted properly
            Func<UserDefinedType, int> custTypeLambda = (UserDefinedType a) => a.a;
            Func<UserDefinedStruct, int> custStructLambda = (UserDefinedStruct b) => b.a-1;
            //float and double types (value types) converted to  Single, Double struct
                types
            Func<float, float> floatLambda = (f) => f-1.3f;
            Func<double, double> doubleLambda = (f) => f-1.3d;
            Func<Single, Single> float2 = s => 2.0f - s;
            /*
             //SOME cases from the whole C# grammar that are not supported by the converter


            //this will result in error (referencing class field):
            Func<int, int> sideEffects = (n) =>
            {
                Console.WriteLine(text);
                Console.WriteLine(n + this.someClassField);
                someClassField++;
                return n % 2;
            };


            //this time class field  anotherClassField is not hidden
            sideEffects = (n) =>
            {
                Console.WriteLine(text);
                Console.WriteLine(n + anotherClassField);
                modifiedClassField = 1;
                return n % 2;
            };

            //but after small modification it can work:
            LambdaCode lam = this;
            sideEffects = (n) =>
            {
                Console.WriteLine(text);
                Console.WriteLine(n+lam.someClassField);
                return n % 2;
            };

            //nested lambda-this should be converted partially
            //Func<int,Func<int>> nested = (b) => () => b*3;

            //recursive lambda - this should be converted partially
           // Func<Func<int, int>, Func<int, int>> factorial = (fac) => x => x == 0 ? 0 : x
                * fac(x - 1);
    */
        }

        private class UserDefinedStruct
        {
            public int a;
        }
    }
}
```

2.10.2 Output

```
using System;

using System.Collections.Generic;

using System.Linq;


namespace lambda_converter.target_code

{

    class LambdaCode

    {

        List<int> ints = new List<int> { 1, 356, 23, 1, 56, 2, 123, 555, 78, 221, 4, 0, 2,
            5, 1 };

        class lambdClass8

        {

            public float methodLambd(Single s)

            {

                return 2.0f - s;

            }

        }

        class lambdClass7

        {

            public double methodLambd(Double f)

            {

                return f - 1.3d;

            }

        }

        class lambdClass6

        {

            public float methodLambd(Single f)

            {

                return f - 1.3f;

            }

        }

        class lambdClass5

        {

            public int methodLambd(UserDefinedStruct b)
        {
            return b.a - 1;
        }
    }
    class lambdClass4
    {
        public int methodLambd(UserDefinedType a)
        {
            return a.a;
        }
    }
    class lambdClass3
    {
        public int methodLambd()
        {
            return 3;
        }
    }
    class lambdClass2
    {
        public string text;
        public string teee;
        public int abba;
        public void methodLambd(Int32 n)
        {
            Console.WriteLine(text);
            Console.WriteLine(n);
            Console.WriteLine(teee + 3 * abba + text);
        }
    }
    class lambdClass1
    {
        public int methodLambd(Int32 m, Int32 n)
        {
            return m - n;
        }
    }
    class lambdClass0
    {
        public bool methodLambd(Int32 m)
        {
            return m % 2 == 0;
        }
    }

    int someClassField = 100;
    int anotherClassField = 3;
    int modifiedClassField = 0;

    class UserDefinedType
    {
        public int a;
        float c;

    }

    public void Method()
    {
        lambdClass0 lambdInst0 = new lambdClass0();

        //SimpleLambdaExpression
        var even = ints.Where(lambdInst0.methodLambd).ToList();

        int[] externalInts = { 1, 3, 5 };
        int[] localInts = { 3, 6, 9 };
        lambdClass1 lambdInst1 = new lambdClass1();

        //ParenthesisedLambdaExpression
        var zipped = localInts.Zip(externalInts, lambdInst1.methodLambd).ToList();

        string text = "result of zipping";
```

```
string teee = "some random text";

int abba = 123;

lambdClass2 lambdInst2 = new lambdClass2();

lambdInst2.text = text;

lambdInst2.teee = teee;

lambdInst2.abba = abba;            //statement lamda with capture

zipped.ForEach(lambdInst2.methodLambd);

lambdClass3 lambdInst3 = new lambdClass3();


Func<int> voidLam = lambdInst3.methodLambd;

lambdClass4 lambdInst4 = new lambdClass4();


//Check if custom reference types converted properly

Func<UserDefinedType, int> custTypeLambda = lambdInst4.methodLambd;

lambdClass5 lambdInst5 = new lambdClass5();
Func<UserDefinedStruct, int> custStructLambda = lambdInst5.methodLambd;

lambdClass6 lambdInst6 = new lambdClass6();
//float and double types (value types) converted to  Single, Double struct
    types

Func<float, float> floatLambda = lambdInst6.methodLambd;

lambdClass7 lambdInst7 = new lambdClass7();
Func<double, double> doubleLambda = lambdInst7.methodLambd;

lambdClass8 lambdInst8 = new lambdClass8();
Func<Single, Single> float2 = lambdInst8.methodLambd;

/*

 //SOME cases from the whole C# grammar that are not supported by the converter




//this will result in error (referencing class field):

Func<int, int> sideEffects = (n) =>

{

    Console.WriteLine(text);

    Console.WriteLine(n + this.someClassField);

    someClassField++;

    return n % 2;

};
```

```
            //this time class field  anotherClassField is not hidden
            sideEffects = (n) =>
            {
                Console.WriteLine(text);
                Console.WriteLine(n + anotherClassField);
                modifiedClassField = 1;
                return n % 2;
            };

            //but after small modification it can work:
            LambdaCode lam = this;
            sideEffects = (n) =>
            {
                Console.WriteLine(text);
                Console.WriteLine(n+lam.someClassField);
                return n % 2;
            };

            //nested lambda-this should be converted partially
            //Func<int,Func<int>> nested = (b) => () => b*3;

            //recursive lambda - this should be converted partially
           // Func<Func<int, int>, Func<int, int>> factorial = (fac) => x => x == 0 ? 0 : x
                * fac(x - 1);
    */
        }

        private class UserDefinedStruct
        {
            public int a;
        }
    }
}
```

2.11 Lambda Transformer code

2.11.1 Program.cs

```
using System;
using System.IO;
using static lambda_converter.LambdaConverter;

namespace lambda_converter
{

    class Program
    {
        const string CODELOCATION = @".\targetCode\LambdaCode.cs";
        const string RESULTLOCATION = @".\targetCode\NonLambdaCode.cs";

        static void Main(string[] args)
        {
            LambdaConverter conv = new LambdaConverter();
            try
            {
                var code = conv.Convert(File.ReadAllText(CODELOCATION));
                var lines = code.Split('\n');
                using (StreamWriter sr = new StreamWriter(RESULTLOCATION))
                {
                    foreach (var line in lines)
                    {
                        sr.WriteLine(line);
                    }
                }

            }
            catch (IOException ex)
            {
                Console.Error.WriteLine("IO Exception-cannot write output to file" + ex.
                    ToString());

            }
            catch (NotSupportedException ex)
            {
                Console.Error.WriteLine("Some tranformations on code are not supported" +
                    ex.ToString());

            }
            catch (UnsupportedCodeTransformationException ex)
            {
                Console.Error.WriteLine(ex.ToString());
            }


        }
    }
}
```

2.11.2 LambdaConverter.cs

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Editing;
using Microsoft.CodeAnalysis.Formatting;
using Microsoft.CodeAnalysis.Text;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace lambda_converter
{
    public class LambdaConverter
    {
        public class UnsupportedCodeTransformationException : Exception
        {
            public UnsupportedCodeTransformationException(string message) : base(message)
            {
            }
        }
        public class ImproperInputCodeException : Exception
        {
            public ImproperInputCodeException(string message) : base(message)
            {
            }
        }

        private static bool isLambda(SyntaxNode node)
        {
            //all kinds of syntaxnodes that are represented by lambda expressions
            switch (node.Kind())
            {
                case SyntaxKind.ParenthesizedLambdaExpression:
                case SyntaxKind.SimpleLambdaExpression:
                case SyntaxKind.AnonymousMethodExpression:
                case SyntaxKind.LetClause:
                case SyntaxKind.WhereClause:
                case SyntaxKind.AscendingOrdering:
                case SyntaxKind.DescendingOrdering:
                case SyntaxKind.JoinClause:
                case SyntaxKind.GroupClause:
                case SyntaxKind.LocalFunctionStatement:
                    return true;

                case SyntaxKind.FromClause:
                    // The first from clause of a query expression is not a lambda.
                    return !node.Parent.IsKind(SyntaxKind.QueryExpression);
            }

            return false;
        }

        const string LAMBDA_CLASS_BASENAME = "lambdClass";
        const string LAMBDA_METHOD_BASENAME = "methodLambd";
        static int instanceIndex = 0;
        static int classIndex = 0;


        public string Convert(string code)
        {
            var workspace = new AdhocWorkspace();
            var projectId = ProjectId.CreateNewId();
            var versionStamp = VersionStamp.Create();
            var projectInfo = ProjectInfo.Create(projectId, versionStamp, "LambdNewProject"
                , "labdaProjName", LanguageNames.CSharp);
            var newProject = workspace.AddProject(projectInfo);
            var document = workspace.AddDocument(newProject.Id, "NewFile.cs", SourceText.
                From(code));


            var tree = document.GetSyntaxTreeAsync().Result;
            var root = tree.GetRoot();
            var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.
                Location);
            var systemCore = MetadataReference.CreateFromFile(typeof(Enumerable).Assembly.
                Location);
            var options = new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary)
                ;
            var comp = CSharpCompilation.Create("LambdaCode")
                .AddReferences(mscorlib, systemCore)
                .AddSyntaxTrees(tree)
                .WithOptions(options);
            var semantic = comp.GetSemanticModel(tree);

            var lambdas = root.DescendantNodes().Where(m => isLambda(m) == true).ToList();


            Dictionary<SyntaxNode, SyntaxNode> replacement = new Dictionary<SyntaxNode,
                SyntaxNode>();
            List<TransformationInfo> transformations = new List<TransformationInfo>();

            foreach (var lambda in lambdas)
            {
                var transInfo = new TransformationInfo();
                transInfo.OriginalLambdaNode = root.DescendantNodes().First(m => m ==
                    lambda);

                var symbol = semantic.GetSymbolInfo(lambda).Symbol;
                var methodSymbol = symbol as IMethodSymbol;


                if (methodSymbol == null)
                {
                    continue;
                }
                if (methodSymbol.ReturnType != null)
                {
                    var returntype = methodSymbol.ReturnType as TypeSyntax;
                    var parsedReturntype = SyntaxFactory.ParseTypeName(methodSymbol.
                        ReturnType.ToDisplayString());

                    var lambdaExpression = lambda as LambdaExpressionSyntax;

                    if (lambdaExpression != null)
                    {
                        BlockSyntax lambdaBody;


                        DataFlowAnalysis result = semantic.AnalyzeDataFlow(lambdaExpression
                            );

                        var captured = result.DataFlowsIn;
                        var parentClass = lambda.Ancestors().OfType<ClassDeclarationSyntax
                            >().First();
                        var parentClassFields = semantic.LookupSymbols(parentClass.
                            ChildNodes().OfType<MethodDeclarationSyntax>().First().SpanStart
                            ).OfType<IFieldSymbol>();
                        var lambdaFields = semantic.LookupSymbols(lambda.SpanStart).OfType<
                            ILocalSymbol>();

                        var paramsListString = "(" + string.Join(", ", methodSymbol.
                            Parameters.Select(m => m.Type.Name + " " + m.Name)) + ")";


                        if (methodSymbol.ReturnType.SpecialType == SpecialType.System_Void)
```

```
{
    //simply copy lambda body
    lambdaBody = SyntaxFactory.Block(
        lambdaExpression.Body.DescendantNodes().OfType<
            StatementSyntax>());

}
else
{
    ExpressionSyntax expr = SyntaxFactory.ParseExpression(
        lambdaExpression.Body.ToFullString());
    if (lambdaExpression.DescendantNodes().OfType<
        ReturnStatementSyntax>().Any())
    {
        //do not insert return statement as it is already present
        lambdaBody = SyntaxFactory.Block(lambdaExpression.Body.
            DescendantNodes().OfType<StatementSyntax>());
    }
    else
    {
        //add return statement
        lambdaBody = SyntaxFactory.Block(SyntaxFactory.
            ReturnStatement(expr));
    }
}

var className = LAMBDA_CLASS_BASENAME + classIndex++;
var instanceName = "lambdInst" + instanceIndex++;


var methodDef = SyntaxFactory.MethodDeclaration(parsedReturntype,
    LAMBDA_METHOD_BASENAME)
.WithParameterList(SyntaxFactory.ParseParameterList(
    paramsListString))
.WithBody(lambdaBody).WithModifiers(SyntaxFactory.TokenList(
    SyntaxFactory.Token(SyntaxKind.PublicKeyword))).
    NormalizeWhitespace().WithTrailingTrivia(SyntaxFactory.EndOfLine
    ("\n"));

var fields = captured.Where(m => (m as ILocalSymbol) != null).
    Select(m =>
{
    var sym = (m as ILocalSymbol);

    var type = SyntaxFactory.ParseTypeName(sym?.Type?.
        ToDisplayString());
    var name = sym?.Name;
    return SyntaxFactory.FieldDeclaration(SyntaxFactory.
        VariableDeclaration(type).WithVariables(SyntaxFactory.
        SingletonSeparatedList(SyntaxFactory.VariableDeclarator(
        SyntaxFactory.Identifier(name))))).WithModifiers(
        SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.
        PublicKeyword)));
});


if (captured.Where(m => (m as IParameterSymbol) != null).Any())
{

    throw new UnsupportedCodeTransformationException("Cannot
        convert lambdas that refer to class fields using this
        keyword. Is this keyword neccessary?");
}

var methods = SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
    methodDef);
SyntaxList<MemberDeclarationSyntax> members = SyntaxFactory.List(
    fields.Union(methods));
```

```
                var classDecl = SyntaxFactory.ClassDeclaration(className)
                                             .WithMembers(members)
                                             .NormalizeWhitespace()
                                             .WithTrailingTrivia(SyntaxFactory.
                                                 EndOfLine("\n"));

                transInfo.ClassDeclaration = classDecl;

                //instantiation and field filling

                var instanceSyntax = SyntaxFactory.LocalDeclarationStatement(
                    SyntaxFactory.VariableDeclaration(
                        SyntaxFactory.IdentifierName(
                            SyntaxFactory.Identifier(className)))
                            .WithVariables(SyntaxFactory.SingletonSeparatedList(
                                SyntaxFactory.VariableDeclarator(
                                    SyntaxFactory.Identifier(instanceName))
                                    .WithInitializer(SyntaxFactory
                                    .EqualsValueClause(SyntaxFactory
                                    .ObjectCreationExpression(SyntaxFactory.
                                        IdentifierName(className))
                                    .WithArgumentList(SyntaxFactory.ArgumentList())
                                        )))))
                     .NormalizeWhitespace().WithTrailingTrivia(SyntaxFactory.
                         EndOfLine("\n")); ;

                //fill each capturing field
                var capturingFieldsAssignments = captured.Where(m => (m as
                    ILocalSymbol) != null).Select(m =>
                {
                    var sym = (m as ILocalSymbol);
                    var type = SyntaxFactory.ParseTypeName(sym.Type.ToDisplayString
                        ());
                    var name = sym.Name;
                    var capturingFieldAssignment = SyntaxFactory.
                        ExpressionStatement(SyntaxFactory.AssignmentExpression(
                        SyntaxKind.SimpleAssignmentExpression, SyntaxFactory.
                        MemberAccessExpression(SyntaxKind.
                        SimpleMemberAccessExpression, SyntaxFactory.IdentifierName(
                        instanceName), SyntaxFactory.IdentifierName(name)),
                        SyntaxFactory.IdentifierName(name)));

                    return capturingFieldAssignment.NormalizeWhitespace().
                        WithTrailingTrivia(SyntaxFactory.EndOfLine("")); ;
                });

                transInfo.InstanceInitSyntax = instanceSyntax;
                transInfo.StatementBeforeLambdaExpression =
                    capturingFieldsAssignments.ToList();

                //method call
                var method = SyntaxFactory.ParseExpression(instanceName + "." +
                    methodDef.Identifier.ToFullString());

                transInfo.MethodUsage = method;

                transformations.Add(transInfo);
            }
        }
    }
    // https://joshvarty.wordpress.com/2015/08/18/learn-roslyn-now-part-12-the-
        documenteditor/
    var documentEditor = DocumentEditor.CreateAsync(document).Result;


    var firstChild = root.DescendantNodesAndSelf().OfType<ClassDeclarationSyntax>()
        .FirstOrDefault()?.DescendantNodes()?.FirstOrDefault();
    if (firstChild == null)
        throw new ImproperInputCodeException("Supplied code is an improper C# code.
                     No parent class.");

            //transform code
            foreach (var trans in transformations)
            {
                documentEditor.InsertAfter(firstChild, trans.ClassDeclaration);
                var statements = trans.OriginalLambdaNode.Ancestors().OfType<BlockSyntax>()
                    .FirstOrDefault().ChildNodes()
                    .OfType<LocalDeclarationStatementSyntax>().ToList<SyntaxNode>().Union(
                        trans.OriginalLambdaNode.Ancestors()
                    .OfType<ExpressionStatementSyntax>().ToList<SyntaxNode>()).ToList();

                var ancestors = trans.OriginalLambdaNode.Ancestors()
                    .OfType<LocalDeclarationStatementSyntax>().ToList<SyntaxNode>()
                    .Union(trans.OriginalLambdaNode.Ancestors()
                    .OfType<ExpressionStatementSyntax>().ToList());
                var index = statements.IndexOf(ancestors.FirstOrDefault());

                var prevStatement = statements.ElementAtOrDefault(index);

                if (prevStatement != null)
                    documentEditor.InsertBefore(prevStatement, (new List<SyntaxNode> {
                        trans.InstanceInitSyntax }).Union(trans.
                        StatementBeforeLambdaExpression));

                documentEditor.ReplaceNode(trans.OriginalLambdaNode, trans.MethodUsage);
            }

            try
            {

                var updatedDoc = Formatter.FormatAsync(documentEditor.GetChangedDocument())
                    .Result;

                string resultCode = updatedDoc.GetSyntaxTreeAsync().Result.ToString();

                return resultCode;
            }
            catch (IOException ex)
            {
                throw ex;
            }
            catch (Exception ex)
            {
                throw new NotSupportedException("Unexpected transformation error occured
                    while generating final output: " + ex.ToString());
            }
        }

    }
}
```

2.11.3 TransformationInfo.cs

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace lambda_converter
{
    class TransformationInfo
    {
        public SyntaxNode OriginalLambdaNode;
        public ClassDeclarationSyntax ClassDeclaration;
        public LocalDeclarationStatementSyntax InstanceInitSyntax;
        public List<ExpressionStatementSyntax> StatementBeforeLambdaExpression;
        public ExpressionSyntax MethodUsage;
    }
}
```

Bibliography

[1] https://en.wikipedia.org/wiki/Lambda_calculus accessed 01.05.2017

[2] R. Rojas, A Tutorial Introduction to the Lambda Calculus

[3] https://msdn.microsoft.com/en-us/library/aa664812(v=vs.71).aspx

[4] https://msdn.microsoft.com/en-us/library/bb397687.aspx accessed 01.05.2017

[5] https://github.com/dotnet/roslyn

[6] https://github.com/dotnet/roslyn/wiki/Roslyn Overview

[7] http://www.amazedsaint.com/2011/10/c-vnext-roslynan-introduction-and-quick.html