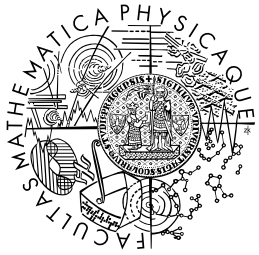


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Petr Houška

Compilation of a dynamic language Generators into MSIL

Department of Software Engineering

Supervisor of the bachelor thesis:	Mgr. Jakub Míšek
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my advisor Mgr. Jakub Míšek for his valuable advice and guidance he has given me for this thesis. I would also like to thank him for starting the Peachpie project in the first place and bringing me on despite the disadvantages mentoring a student who is writing a bachelor thesis inherently brings.

This thesis would also not be possible without the endless support of my parents, friends, and classmates and the endless encouragement of my girlfriend. All of them and practically everyone else who I had the pleasure to meet during my studies deserve an acknowledgment.

Title: Compilation of a dynamic language Generators into MSIL

Author: Petr Houška

Department: Department of Software Engineering

Supervisor: Mgr. Jakub Míšek, Department of Software Engineering

Abstract: The goal of this thesis is to design and implement support for generators within the Peachpie framework, a PHP to CIL compiler. Generators are the simplest form of methods that resume from the same state in which they returned earlier when called repeatedly. The reference PHP interpreter Zend engine supports generators natively. Due to that fact generators in PHP are support a number of features not usually seen in other languages. CIL on the other hand does not have a native support for generators. Therefore, languages build on top of CIL (e.g. C#, F#) have to implement them by other means such as by rewriting the original generator methods into state machines. In this thesis we will design and implement support for generators through semantic tree transformations. All that with the intention of keeping the maximum possible compatibility with reference PHP generators. We will also make a comparison to generators in C# whose main implementation also uses CIL as a backend.

Keywords: compiler php msil .net generators roslyn peachpie

Contents

Introduction

1. Title of the first chapter

An example citation: Anděl [2007]

1.1 Title of the first subchapter of the first chapter

Now that all external components can handle bound bags with pre-bound blocks it is time to actually implement the algorithm that creates them. The implementation is based on one core principle.

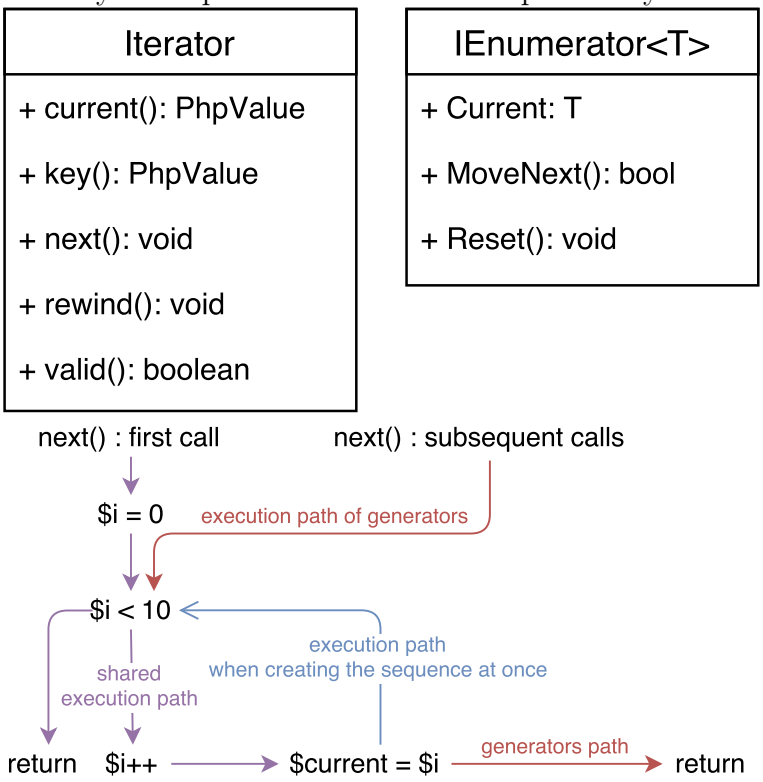
When the semantic binds an expression that is, in the current expression tree, under a path between a yield and the root, it does not return a bind version of it. Instead it creates its bound representation, puts an assignment statement that sets said bound expression to a new temporal variable into a before current tree block, and returns a read from the new temporal variable. That way the temporal variable is put in the expression tree instead of the original expression. Then when it is supposed to return the whole bound tree it returns both it, with some branches replaced with temporal variables, and the prebound statements in a bound bag.

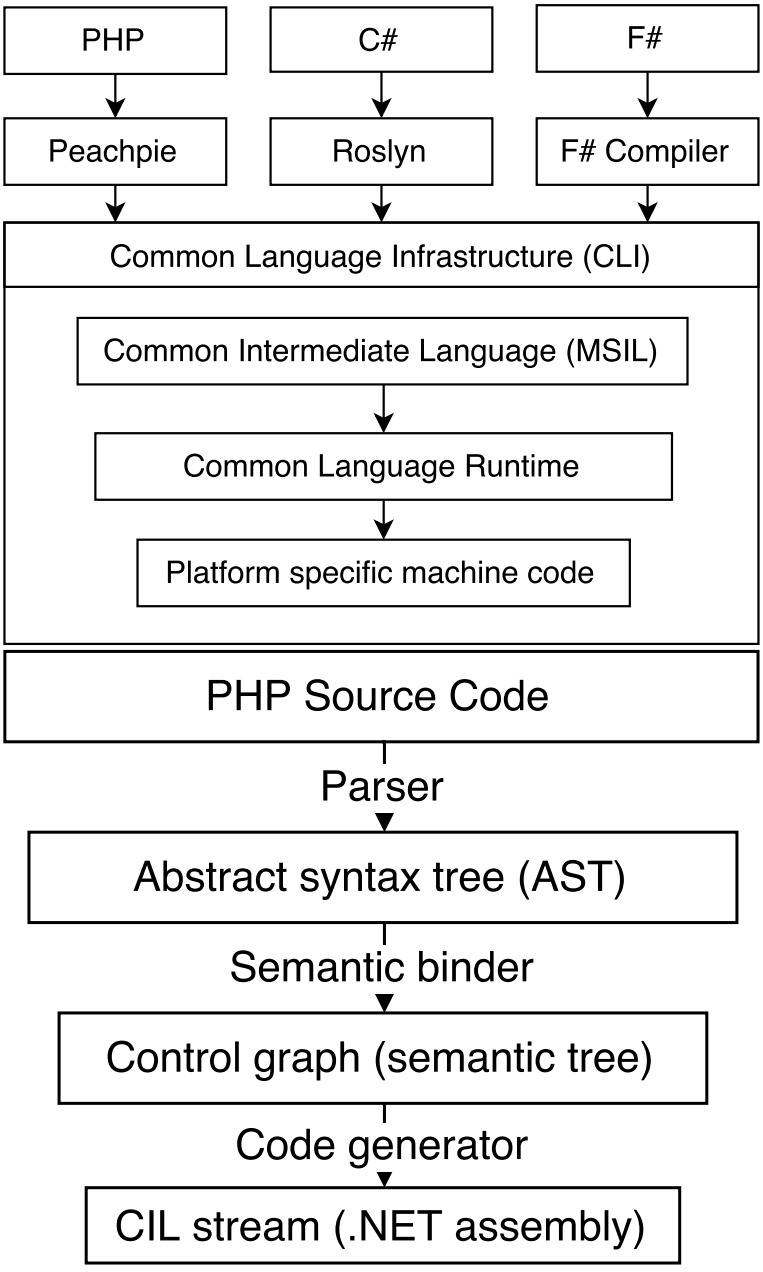
```
.method public hidebysig
instance void M (
int32 a
) cil managed {
.maxstack 4
.locals init (
[0] int32,
[1] int32)
IL_0000: nop                    // Do nothing (No operation)
IL_0001: ldc.i4.3               // Push 3 onto the stack as int32
IL_0002: stloc.0                // Pop a value from stack into local
 ↪  variable 0
IL_0003: ldc.i4.5               // Push 5 onto the stack as int32
IL_0004: stloc.1                // Pop a value from stack into local
 ↪  variable 1
IL_0005: ldarg.0                // Load argument 0 (this) onto the
 ↪  stack
IL_0006: ldarg.1                // Load argument 1 onto the stack
IL_0007: ldloc.0                // Load local variable 0 onto stack
IL_0008: ldloc.1                // Load local variable 1 onto stack
IL_0009: mul                    // Multiply values
IL_000a: add                    // Add two values, returning a new
 ↪  value
IL_000b: call instance int32 C::G(int32) // Call method indicated
 ↪  on the stack with arguments
IL_0010: pop                    // Pop value (returned from G) from
 ↪  the stack
```

```
IL_0011: ret                    // Return from method, possibly with
 ↪  a value
} // end of method C::M
```

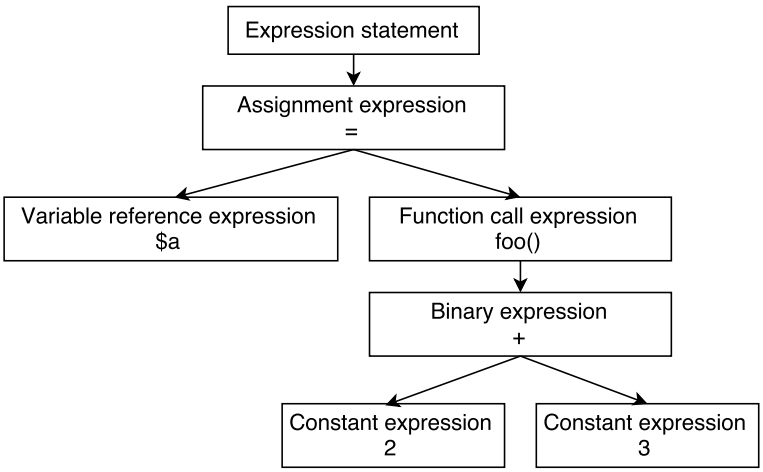
1.2 Title of the second subchapter of the first chapter

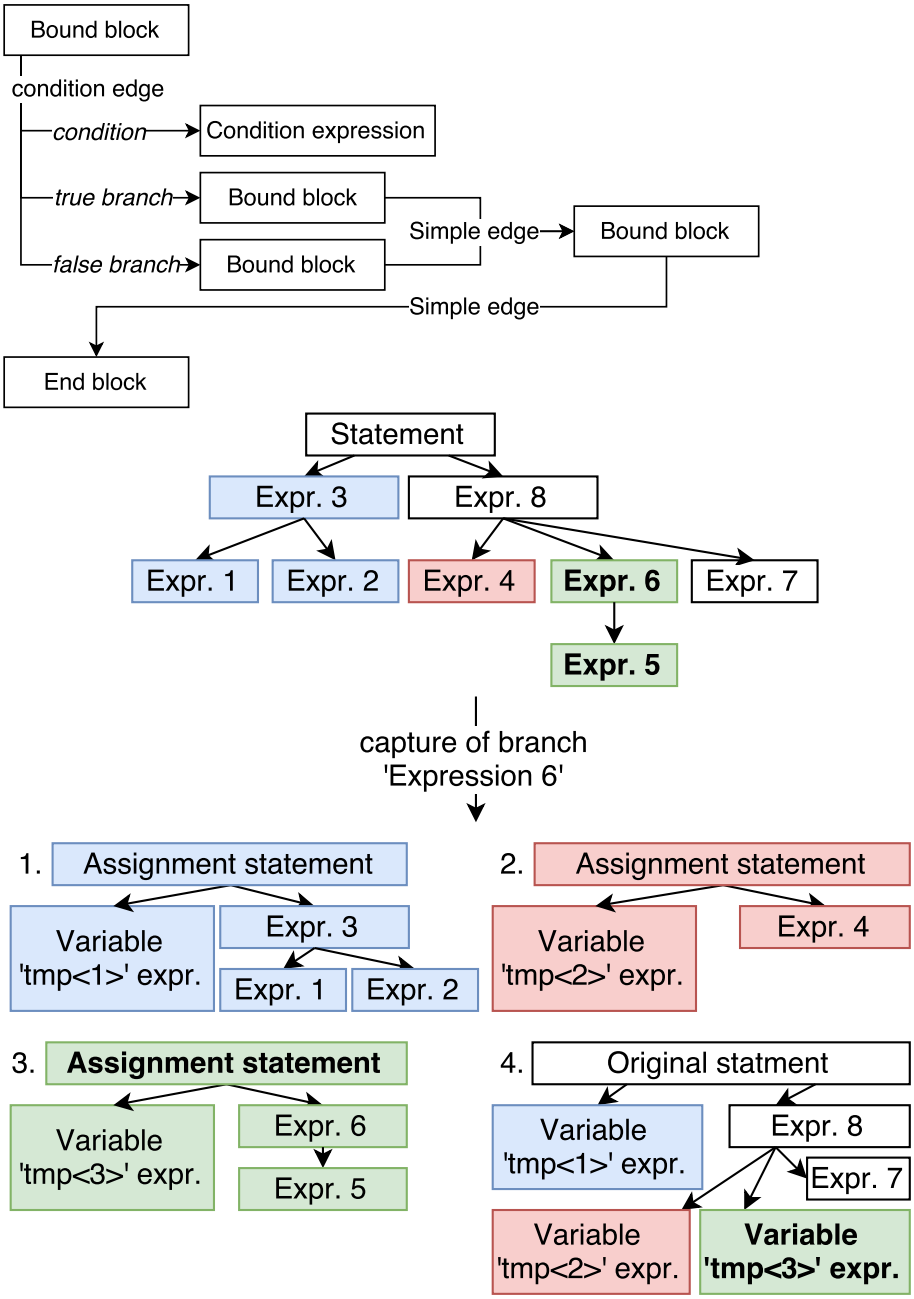
There are two important observations required for this method. First, a yield can be broken into two semantic nodes. A statement that does the value and key setting, state saving, return, and marking the continuation label. It is basically an equivalent of C# yield. And an expression that represents the sent value. If the expression directly follows the statement the result is the same as with one combined yield expression that was used previously.




PHP
C#
F#
Peachpie
Roslyn
F# Compiler
Common Language Infrastructure (CLI)
Common Intermediate Language (MSIL)
Common Language Runtime
Platform specific machine code
PHP Source Code
Parser
Abstract syntax tree (AST)
Semantic binder
Control graph (semantic tree)
Code generator
CIL stream (.NET assembly)

$a = foo(2+3);


Expression statement
Assignment expression
=
Variable reference expression
$a
Function call expression
foo()
Binary expression
+
Constant expression
2
Constant expression
3



Second, we can take any branch from an expression tree and prepend it before the tree while keeping the meaning of the program same except for the order of execution. To do it we need to create a temporal variable, replace the branch in the tree with a read from said variable, and prepend the tree with a statement that assigns the branch that was replaced to the variable it was replaced with. Let us call this process capturing a branch.

The problem with the order of execution is that the captured branch, being lifted to the prepended statement, will get executed before any other expression from the tree. Even before all the expressions in branches that might be to the left from the captured branch and that were therefore supposed to be executed first.

An obvious solution to this problem is to take and prepend not only the one branch we want to capture but also, in their respective left to right order, all other branches that are supposed to get executed before it. Since the semantic

graph emit and thus also execution follows a post-order traversal that means all branches that are higher and to the left from the branch we want to capture.

Specifically, all branches that start to the left from the path between the root of our captured branch and the root of the whole semantic graph. All other expressions, be it those directly on the path or on branches to the right, are supposed to be evaluated after our branch and as such do not have to be taken and prepended. The ones on the path have our branch among their children and thus needs its result first to be evaluated. And the ones on right need to be evaluated later simply because of post-order traversal rules.

1.2.1 Title of the second subchapter of the first chapter

In Peachpie the actual CIL code generation is based solely on the semantic tree data structure. Each of its elements, be it an edge or a statement in a form of an expression tree, contains a method that is able to produce its representation in CIL.

Title of the second subchapter of the first chapter

For now let us suppose that there is only one yield in a method, that it is part of some larger expression tree, and that it has not yet been splitted into a statement and an expression.

For now let us suppose that there is only one yield in a method, that it is part of some larger expression tree, and that it has not yet been splitted into a statement and an expression.

For now let us suppose that there is only one yield in a method, that it is part of some larger expression tree, and that it has not yet been splitted into a statement and an expression.

For now let us suppose that there is only one yield in a method, that it is part of some larger expression tree, and that it has not yet been splitted into a statement and an expression.

2. Title of the second chapter

2.1 Title of the first subchapter of the second chapter

2.2 Title of the second subchapter of the second chapter

Conclusion

3. Attachments

Attached to this thesis is a snapshot of Peachpie project's git repository. It contains not only the implementation that was done as the practical part of this thesis but also the rest of the complete project. A more up to date version can be found on github[1].

To query only commits done by the author of this thesis please filter out author *Petr Houška* or email *houskape@gmail.com*.

3.1 Compilation

The project's only implicit dependency is .NET Core runtime and optionally its CLI SDK. If you want to compile the project yourself you can download both of them from the official site[2], for Linux, Windows, or MacOSX.

After obtaining the .NET Core SDK please navigate to the folder with the Peachpie repository in your favourite terminal and:

```
dotnet restore  //download all external packages required
dotnet build    //build the complete solution
```

3.2 Structure

There are three components relevant for this thesis within the repository. The compiler binaries, the compiler implementation, and the generators tests. Below are listed paths to them and in case of the compiler's implementation also to some files containing the majority of our work to support generators.

1. src/Compiler/peach
2. src/CodeAnalysis
   (a) ./Semantics/SemanticsBinder.cs
   (b) ./Semantics/Graph/BuilderVisitor.cs
   (c) src/Peachpie.Runtime/std/Generator.cs
3. tests/generators

3.3 Manual testing

To compile an arbitrary PHP file into a .NET assembly with Peachpie invoke the compiler with a path to the PHP file as its first argument. The compiler assembly resides at aforementioned path and is called peach.exe or peach.dll depending of whether it was compiled for full .NET framework or .NET Core.

[1] github.com/peachpiecompiler/peachpie
[2] microsoft.com/net/download/core

```
$\src\Compiler\peach> dotnet run .\test.php
```

Please do note that an assembly compiled this way will require PHP runtime libraries to run. These libraries can be found, for example, in the bin output of the compiler (peach) project.

Alternatively it is possible to use a Peachpie console application sample[3]. It includes a .msbuildproj file that configures the .NET Core CLI to download and use both the Peachpie compiler toolchain and required runtime libraries automatically. More about that approach can be found on a peachpie blog[4].

3.4 Automatic testing

The Peachpie project includes a comprehensive set of automatic tests. These consist of PHP files that get compiled by the Peachpie compiler and run by a .NET runtime. If there is a PHP runtime present in the current path environment variable they get run by it as well. The results are then compared to ensure Peachpie compilation keeps the original PHP semantics and is, in terms of runtime behaviour, indistinguishable from the reference implementation.

There is a number tests created as part of this thesis that ensure the implementation of generators support works correctly. They are located in a subfolder tests/generators. While they are in no particular order it is generally true that the higher their number the more complex aspect of generators they test. Below is a command that invokes all peachpie tests, including generator ones.

```
$\src\Tests\Peachpie.ScriptTests> dotnet test
```

Please do note that two tests usually fail on some machines because of encoding issues.

[3] github.com/iolevel/peachpie-samples/tree/master/console-application
[4] peachpie.io/2017/04/tutorial-vs2017.html

Bibliography

J. Anděl. *Základy matematické statistiky.* Druhé opravené vydání. Matfyzpress, Praha, 2007. ISBN 80-7378-001-1.

List of Figures

List of Abbreviations

CLI Common language infrastructure, open standard for runtime environment implemented by .NET, Mono, and others.

CIL Common intermediate language, object oriented assembler defined by *CLI* (also known as MSIL or IL).

CLR Common language runtime, virtual machine implementing the execution engine specified by *CLI*.

DLR Dynamic language runtime, set of libraries providing compiler and runtime services for dynamic languages build on top of *CLR*.

AST Abstract syntax tree, structured representation of the source code.

CFG Control flow graph, a semantic graph representing a method.