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 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

Despite a slow decline in recent years [TIOBE, 2017], PHP is still one of the main languages used for a server side programming on the web [Stack Overflow, 2017]. Its only two relevant implementations are the reference and almost exclusively used Zend engine[1] and slowly emerging HHVM by Facebook[2]. Both of them are standalone virtual machines and neither of them supports easy interfacing with the outside world. Hence, it is quite difficult to share code between a web backend and, for example, a mobile or traditional desktop application.

Fortunately, there is a solution in the form of a Peachpie project[3] that is being researched at the Charles University. The project aims to provide a compiler from PHP to ".NET bytecode" CIL[4] and a reimplementation of PHP base class library, thus creating a bridge between PHP and the whole .NET ecosystem. Due to it being a full compiler that takes PHP sources and spits out .NET assemblies indistinguishable from those created by other .NET languages compilers (e.g C#, F# or IronPython) it provides both ways interoperability. It enables both calling normal unmodified .NET libraries from PHP and vice versa. Also, thanks to an extensive compile-time type analysis and proven .NET just in time compiler (RiuJIT) it achieves better performance than reference Zend engine in certain operations [Míšek, 2017], [Fistein and Míšek, 2016 - 2017].

PHP, like many other modern languages, has a first class support for generators. Simply put, generators are methods that resume computation from the very place and with the same state they returned at previously when called repeatedly. They are usually used for generating large sequences of data lazily, hence the name generators. Since the execution state gets saved automatically on the special pause and return places (usually called yields) one can write an algorithm as if the sequence was being created at once and only insert yields at appropriate times, e.g. when a new item gets created. The language handles the rest. Each subsequent call to the generator method resumes computation from the last evaluated yield and continues to the next one, e.g. creating a new element each time.

The Zend engine has a native support for generators. It intrinsically understands yields and is, on their evaluation, able to save the state of current execution [Popov, 2017]. CIL has no such first class support. For that reason languages built on top of CIL have to implement generators through other means [Lippert, 2008]. Usually by rewriting generator methods into state machines with explicit state saving before each yield and state retrieval in the beginning.

That is exactly what this thesis covers. It describes the design and implementation of support for PHP generators within the Peachpie compiler through semantic tree transformations, implementation of new semantic tree nodes, and extensions to Peachpie runtime library. In the implementation parts it tries to not only plainly cover the code but also to depict the decision process that led to choosing certain approaches over others. During the whole work we will compare

[1]zend.com/en/community/php

[2]hhvm.com/

[3]peachpie.io/

[4]Chapter 2.1

our approach with the one taken by C# team and its compiler Roslyn. C# was chosen as a reference language due to it being the prominent language in .NET platform.

While the goal is to implement support for generators with as much original PHP semantic as possible, due to the scope of this work we will not discuss the specific implementation of all PHP generators features. Namely, we will not cover handling yields in exception control blocks (try, catch, finally) in detail and will leave its implementation for future work.

## Thesis structure

This thesis is divided into seven chapters. The first one covers general concepts of generators both in PHP and in other languages, explaining what they are, what features and limitations do they have, and where they stand in regards to iterators.

The second chapter briefly introduces the .NET platform and its intermediate language CIL. The third is all about the Peachpie project. It describes its architecture focusing mainly on the semantic tree data structure and CIL emit phase of the compiler. In fourth chapter we take a look at how generators are implemented in C#'s Roslyn and PHP's Zend engine. Especially the Roslyn's approach is important because it serves as a basis for our own implementation.

Generators within Peachpie is the focus of the fifth chapter which itself is further divided into five more sections. First describes an implementation of generators limited to circa C# generators. It builds on the theoretical basis described in previous section about Roslyn's approach. Second proposes theoretical algorithm to handle yield as expression. Third talks about implementation of said algorithm within peachpie. Fourth briefly mentions possible solutions for yields in exception handling blocks. And the fifth is about possible future work that could be done for generators support within peachpie.

The sixth chapter concludes and summarizes the whole thesis. And last but not least the final chapter provides a lightweight user documentation for the peachpie project and overview of attachments.

# 1. Generators

## 1.1 Iterators

Before going into generators we need to define iterators first. Iterators, as their name suggests, represent a state of iteration of some sequence backed by either themselves or some other object. In both C# and PHP they can be arbitrary objects implementing an *Iterator*[1] (for PHP) or an *IEnumerator*[2] (in C#) interface.

Both of these contain a number of methods (Figure 1.1). However, for now we will focus on only two of them they both share, *current* and *next*. The first one - *current* - should always return the element the iterator is currently pointing at. As such it should never modify the state of the iterator and should generally be side effect free. The second one, *next*, should advance the iterator to the next item effectively changing what element the *current* method returns.

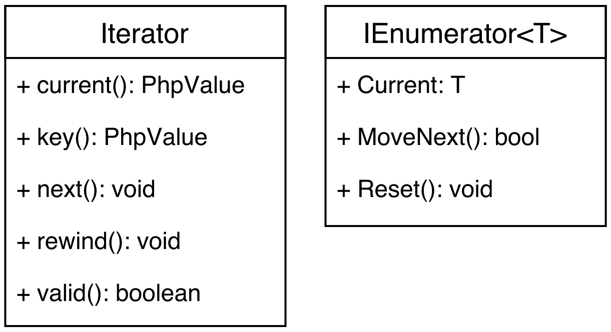

Figure 1.1: Iterator and IEnumerable interfaces.

Iterators serve, among other things, as a useful tool to handle large sequences of items. Instead of generating everything at once and then keeping it, for example, in an array one can simply create an iterator that generates and serves the elements one by one. This enables lower memory footprint because there is no not need to ever keep all the elements in memory at once.

Furthermore instead of spending a lot of resources at once in the beginning to create the whole sequence the iterator can now spend just a little bit for each element's creation when the *next* method is called. Though the final price is the same regardless it enables a more even distribution of the performance hit. Also, one does not need to pay for the elements that do not get created - the ones that are not iterated over.

There is one problem with iterators, though. They are quite tedious to write. The main issue stems from the fact that unlike in a normal method in which you would create the whole sequence at once in the iterator's *next* method you need to always explicitly save and then retrieve the current state of the execution. There is a lot of boilerplate associated with them as well. You need to create a new type and implement a number of methods that do not actually do anything useful.

[1][PHP.Net]
[2][MSDN]

To give an example, creating an array of numbers 1 to $n$ is a straightforward for loop with an assignment. In the case of an iterator's *next* method you need to first retrieve last used number, increase it by one, and then store it. You also need to implement a *current* method that returns the last stored index and a few other ones that are basically just a busywork.

```php
<?php
function by_one_at_once(){
  $result = [];
  for($i = 0; $i < 10; $i++){
    $result[] = $i;
  }
  return $result;
}

class byOneIterator implements Iterator {
  private $position = 0;

  public function rewind() {$this->position = 0;}
  public function key() {return $this->position;}
  public function current() {return $this->position; }
  public function next() {
  $curr_pos = $this->position;
    $curr_pos += 1;
    $this->position = $curr_pos;
  }
  public function valid(){ return ($this->position < 10); }
}
```

While for a monotone sequence of numbers even the iterator is still simple and readable it quickly stops to be the case with higher complexity of the iteration. The amount of code needed for state keeping increases quite quickly and an algorithm which would otherwise be really straightforward when used for the creation of the whole sequence at once gets convoluted. And that is where generators come into play.

## 1.2 Generators universally

Generators provide an easy way to write methods that return iterators while the code can be almost the same as if they returned whole sequences at once instead. All the transforming of the algorithm into the *next* method with its retrieving of the last state in the beginning and state saving after a new element gets set as *current* is handled transparently for the programmer. There is also no need to create a new type and implement other iterator's busywork methods with them. They automatically return correct and fully implemented iterators.

To achieve this a new keyword *yield* or *yield return* is usually introduced. It serves two purposes. First, it marks the spots where the *next* method of the returned iterator should stop executing and save the current state. Second, much like the *return* keyword it specifies a value being returned, in this case actually a value that should be set as *current* on the iterator.

To continue with our example of creating a sequence of numbers from 1 to $n$. One would write a generator method the same way a normal method generating the whole array with the only difference that instead of an assignment into a result array the for loop would contain a *yield* of the current index.

```php
<?php
function by_one_generator(){
  for($i = 0; $i < 10; $i++){
    yield $i;
  }
}
```

An iterator returned from such generator method (Figure 1.2) would have a *next* method that would always start from the last encountered *yield*, execute update and condition part of the for loop, and then set a new element as the *current* one.

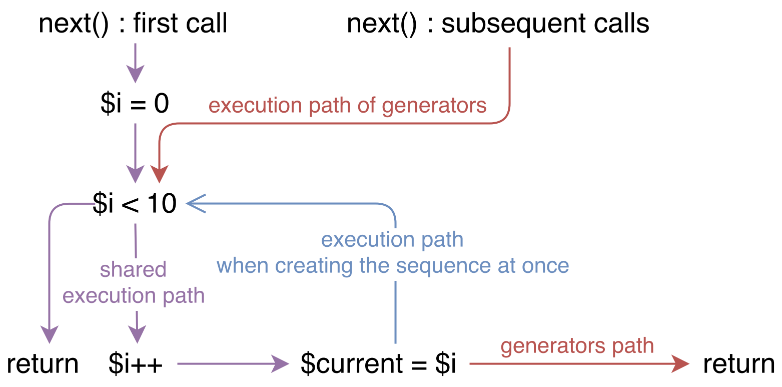

Figure 1.2: Generator's control flow graph.

## 1.3 Generators in PHP

Generators in PHP are not just about creating data, however. They support consuming data as well. Namely, one can send an arbitrary value into the returned iterator through its method called *send*. This method takes one argument - the value being sent, sets it as a result of the last *yield* expression, and resumes the evaluation the same way as *next* method would.

```php
<?php
function logger_generator(){
  $logger = new Logger();
  while(($line = yield) != "END"){
    $logger->log($line);
  }
  $logger->close();
}
```

```
class Logger{
  public function close(){ echo "END;";}
  public function log($line){ echo $line; }
}
```

This means that unlike in C# where *yield* is a statement and therefore can not be a part of a bigger computation or a function call in PHP *yield* can be literally anywhere, even in the place of a function call argument. This further complicates the state saving. In addition to all the local variables when *yield* is part of some bigger expression we need to save its state as well. And it needs to be done the right way to ensure correct order of execution.

Due to various design reasons [Lippert, 2009b] C# also limits where *yields* can happen in regards to exception control blocks. They are not allowed in *catch*[3] and *finally* blocks altogether and can be only in *try* blocks[4] that do not have any associated catch blocks. PHP, on the other hand, allows yielding everywhere, be it in *try*, *catch*, or *finally* blocks.

## 1.4 Generators in other languages

These limitations are not unique to C#, however. Both F# and Visual Basic, the only other truly mainstream CIL based languages, also support generators with these restrictions. For them *yield* is just a statement that can not appear in certain exception handling blocks.

In fact, *yield* being an expression holding a send value is not even a feature all dynamic languages share. JavaScript, for example, had just a brief support[5] for such behavior and as of ECMA 2015 has *yield* only as a statement.

That, nonetheless, does not mean that PHP is completely unique in regards to generators. In python[6], another mainly interpreted dynamic language, *yields* are expressions and are allowed in exception handling blocks almost the same way as in PHP. In Lua[7] they are implemented as a special version of coroutines and thus have an even wider set of features.

[3][Lippert, 2009a]
[4][Lippert, 2009c]
[5][MDN]
[6][docs.python.org]
[7][Lua.org]

# 2. .NET platform

## 2.1 Common intermediate language

### 2.1.1 Evaluation stack

### 2.1.2 Exception handling

# 3. Peachpie project

## 3.1 Peachpie architecture

## 3.2 Peachpie compiler

## 3.3 Semantic graph

### 3.3.1 Statements and expressions

### 3.3.2 Graph structure

### 3.3.3 Graph creation

## 3.4 CIL emit phase

### 3.4.1 Code generator

### 3.4.2 Emit

### 3.4.3 Generate methods' invariants

## 3.5 Peachpie runtime library

# 4. Generators implementation in other platforms

## 4.1 CSharp and Roslyn

### 4.1.1 Iterator object and generator methods

### 4.1.2 Rewriter

### 4.1.3 Local variables parameters

### 4.1.4 Execution position

## 4.2 PHP and Zend Engine

# 5. Generators in Peachpie

## 5.1 Basic generators implementation

### 5.1.1 Iterator object

### 5.1.2 Next method implementation and local variables

### 5.1.3 Accessibility of fields on the Generator type

### 5.1.4 Context handling

### 5.1.5 Rewriter

### 5.1.6 Bound yield expression

### 5.1.7 Start block

### 5.1.8 Method symbol

## 5.2 Yield as an expression - theory

### 5.2.1 Possible approaches

### 5.2.2 Branch capture & yield splitting

### 5.2.3 Semantic tree transformation

### 5.2.4 Short circuit evaluation

## 5.3 Yield as an expression - implementation

### 5.3.1 Binding multiple elements

### 5.3.2 Capturing branches with yields

### 5.3.3 Correctness of modified capturing algorithm

### 5.3.4 Creating and keeping the pre-bound graph

### 5.3.5 Path between the root and yields

### 5.3.6 Conditioned branches

### 5.3.7 Implementation remarks

## 5.4 Yield in exception handling blocks

### 5.4.1 Yields and exception handling blocks in PHP

### 5.4.2 Solution in Peachpie

## 5.5 Future work

# Conclusion

# 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*.

## 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
```

## 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

## 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].

## 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

docs.python.org. 6.2.9. yield expressions. URL https://docs.python.org/3/reference/expressions.html#yield-expressions. Accessed: 2017-06-25.

Benjamin Fistein and Jakub Míšek. Peachpie benchmarks, 2016 - 2017. URL http://www.peachpie.io/2017/06/optimizing-php-code-1.html/. Accessed: 2017-06-25.

Eric Lippert. Iterator block implementation details: auto-generated state machines, 2008. URL http://csharpindepth.com/Articles/Chapter6/IteratorBlockImplementation.aspx. Accessed: 2017-06-25.

Eric Lippert. Iterator blocks part four: Why no yield in catch?, July 2009a. URL https://blogs.msdn.microsoft.com/ericlippert/2009/07/20/iterator-blocks-part-four-why-no-yield-in-catch/. Accessed: 2017-06-25.

Eric Lippert. Iterator blocks, part three: Why no yield in finally?, July 2009b. URL https://blogs.msdn.microsoft.com/ericlippert/2009/07/16/iterator-blocks-part-three-why-no-yield-in-finally/. Accessed: 2017-06-25.

Eric Lippert. Iterator blocks, part five: Push vs pull, July 2009c. URL https://blogs.msdn.microsoft.com/ericlippert/2009/07/23/iterator-blocks-part-five-push-vs-pull/. Accessed: 2017-06-25.

Lua.org. 9.1 – coroutine basics. URL https://www.lua.org/pil/9.1.html. Accessed: 2017-06-25.

MDN. Generator. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator. Accessed: 2017-06-25.

Jakub Míšek. Optimizing php code with peachpie – part 1, June 2017. URL http://www.peachpie.io/2017/06/optimizing-php-code-1.html/. Accessed: 2017-06-25.

MSDN. IEnumerator Interface. URL http://php.net/manual/en/class.iterator.php. Accessed: 2017-06-25.

PHP.Net. The Iterator interface. URL http://php.net/manual/en/class.iterator.php. Accessed: 2017-06-25.

Nikita Popov. PHP 7 virtual machine, April 2017. URL http://nikic.github.io/2017/04/14/PHP-7-Virtual-machine.html#generators/. Accessed: 2017-06-25.

Stack Overflow. Stack overflow developer survey results 2017, 2017. URL https://insights.stackoverflow.com/survey/2017#technology/. Accessed: 2017-06-13.

TIOBE. Tiobe index, 2017. URL https://www.tiobe.com/tiobe-index. Accessed: 2017-06-13.

# 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. |