# PHP Generators in .NET PEACHPIE Compilation of a dynamic langauge Generators into MSIL



Petr Houška | Mgr. Jakub Míšek | Department of Software engeneering | github.com/peachpiecompiler/peachpie | github.com/petrroll/bachelor-thesis

#### Goals

- Generators support within Peachpie
- Keeping reference PHP semantics
- Reusing already existing facilities
- Both design and implementation

## Peachpie

- Bridge between PHP and .NET
- PHP to MSIL (CIL) compiler
- Reimplementation of PHP class library
- Written in C#, .NET foundation member

#### Original generator method Generator's next method + RntLib.GetLocals(generator)['locVarA'] + local variable A + RntLib.GetLocals(generator)['locVarB'] + local variable B rewriter + RntLib.GetThis(generator).C + reference to this.C + RntLib.GetLocals(generator)['argD'] + reference to argD Original generator method Generator + next(): void 1: var generator = new <1>GeneratorType() 2: generator.originalThis = this + current(): PhpValue 3: generator.locals = new PhpArray() 4: generator.locals['argD'] = argD + locals: PhpArray 5: return generator + origThis: <respective type> + state: integer + nextMethod: NextMethodDelegate ----+ ctx: Contex

#### MSIL

- Intermediate language for .NET platform (C#, F#, ...)
- High level, stack based, object oriented assembler
- Native understanding of exception handling
- No concept of generators or execution state saving

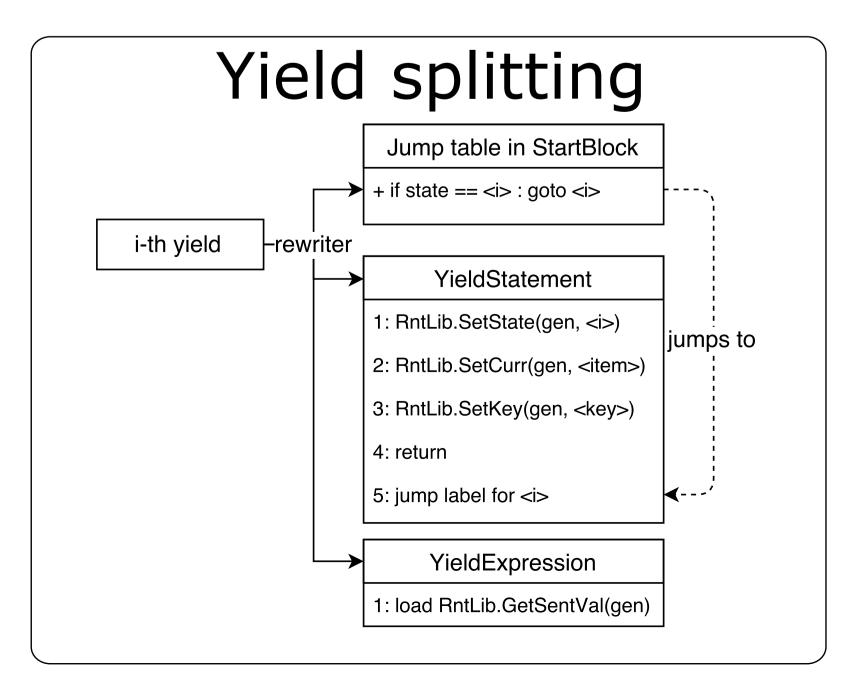
### Generators

- Contain a sequential alghoritm, return an Iterator
- Natively supported by the reference PHP runtime (Zend)
- Lowered by compilers in .NET languages (e.g. Roslyn)
- Execution pauses at *yield* keyword, continues on *next()* call
- Statement in most .NET languages (C#, F#)
- Value carying expression in PHP

#### Generator method next(): first call next(): subsequent calls \$i = 0 execution path of generators -\$i < 10 **←** execution path when creating the sequence at once execution path return \$i++ ---

# Generator method

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



## Solution

- New Generator type representing the returned Iterator
- Changes to the original generator method's body:
- Transformed to Generator's state machine as a new method
- Replaced with code that initiates and returns the Generator
- State machine transformations:
- Local variables lifted to Generator instance
- Exlicit state saving on each yield
- State backed jump table in the beginning
- Semantic tree transformation to lower yields:
- Moves yields under expression trees' roots - Handles conditioned branches with yields
- Maintains the original order of execution
- Splitting yield into a statement and an expression

#### Semantic tree transformation Expr. 5 Expr. 1 Yield Expr. Expr. 4 Expr. 2 capture of yield branch **YieldStatement** Assignment statement Expr. 1 directly followed by Variable 'tmp<1>' expr. Assignment statement **Assignment statement** YieldExpression Expr. 4 Variable Variable 'tmp<2>' expr. 'tmp<3>' expr. Expr. 2 Original statment Assignment statement Variable Expr. 5 Variable 'tmp<4>' expr. 'tmp<4>' expr. Variable Variable 'tmp<1>' expr. 'tmp<3>' expr. **Variable** 'tmp<2>' expr.