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. Benjamin Fistein deserves recognition for his extensive proofreading of this thesis as well.

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 the 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 that generators in PHP support a number of features that are not common in other languages. CIL, on the other hand, does not have a native support for generators. Therefore, languages built 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 the support for generators through semantic tree transformations. All this is handled 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 slight decline in recent years [TIOBE, 2017], PHP is still one of the main languages used for 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 the 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 the 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 the PHP base class library, thus creating a bridge between PHP and the whole .NET ecosystem. Due to the fact that it is a full compiler that takes PHP sources and outputs .NET assemblies indistinguishable from those created by other .NET languages compilers (e.g C#, F# or IronPython), it provides a both-way interoperability. It enables both calling normal unmodified .NET libraries from PHP and vice versa. Also, thanks to an extensive compile-time type analysis and the proven .NET just in time compiler (RiuJIT) it achieves better performance than the 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. In short, generators are methods that, when called repeatedly, resume the computation from the very place and with the same state they returned at previously. 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 were 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 the 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 the CIL have to implement generators through other means [Lippert, 2008] - usually by rewriting generator methods into state machines with the explicit state saving before each yield and a state retrieval in the beginning.

This thesis describes the design and implementation of a support for PHP generators within the Peachpie compiler through semantic tree transformations, an implementation of new semantic tree nodes, and extensions to the Peachpie runtime library. In the implementation parts, the thesis not only tries to plainly cover the code, but also to depict the decision process that led to choosing certain approaches over others. Throughout the work, we will compare our approach

[1]zend.com/en/community/php
[2]hhvm.com/
[3]peachpie.io/
[4]Chapter 2.1

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

While the goal is to implement a 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 generator 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 they have, and where they stand with regards to iterators.

The second chapter briefly introduces the .NET platform and its intermediate language CIL. The third is entirely about the Peachpie project. It describes its architecture, focusing mainly on the semantic tree data structure and CIL emit phase of the compiler. In the fourth chapter, we examine how generators are implemented in C#'s Roslyn and PHP's Zend engine. Roslyn's approach is particularly 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 subsections. The first part describes an implementation of generators limited to circa C# generators. It builds on the theoretical basis described in the previous section about Roslyn's approach. The second one proposes a theoretical algorithm to handle yield as an expression. The third subsection discusses the implementation of said algorithm within Peachpie. In the fourth part, we briefly mention the possible solutions for yields in exception handling blocks. Finally, the fifth subsection is about possible future work that could be done for generator support within Peachpie.

The sixth chapter concludes and summarizes the whole thesis. Ultimately, the final chapter provides a lightweight user documentation for the Peachpie project and an overview of attachments.

1. Generators

1.1 Iterators

Before getting 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 only on two of them that 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 free of side effects. The second one - *next* - should advance the iterator to the next item, effectively changing what element the *current* method returns.

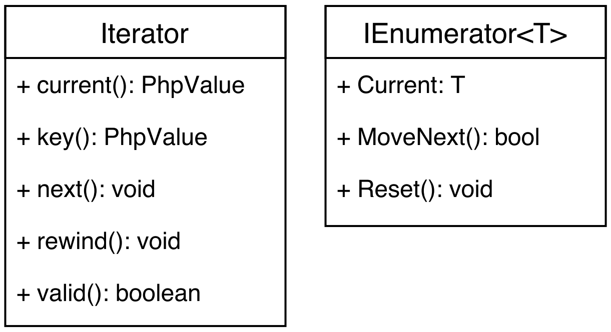


Figure 1.1: Iterator and IEnumerable interfaces.

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 a lower memory footprint, because there is never need to 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 considerably fewer 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, however. They are fairly 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 always need to 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, b]
[2] [MSDN]

To give an example (Listing 1), 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 the 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 essentially 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); }
}
```

Listing 1: Method that creates everything at once and as an Iterator.

While for a monotone sequence of numbers even the iterator is still simple and readable, this quickly ceases to be the case with a 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, becomes convoluted. This is where generators emerge as a relevant solution.

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* (Listing 2), one would write a generator method the same way as a normal method generating the whole array, the only difference being 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;
  }
}
```

Listing 2: By one sequence as a generator.

An iterator returned from such a 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.

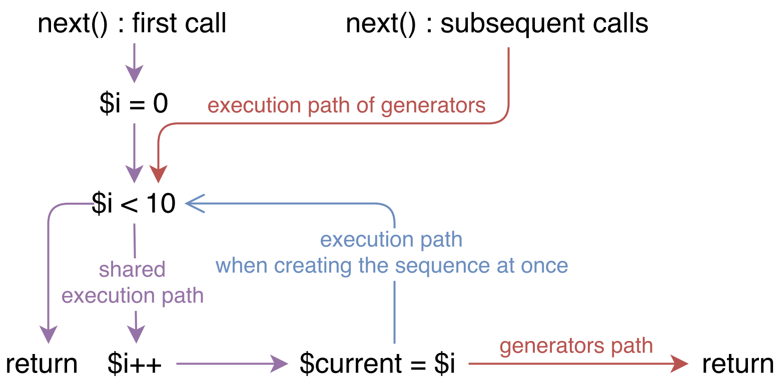


Figure 1.2: Generator's control flow graph.

1.3 Generators in PHP

Generators in PHP are not just about creating data, however. They support consuming data as well (Listing 3). 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 the *next* method would.

```php
<?php
function logger_generator(){
  $logger = new Logger();
  // The sent value is assigned to the $line variable.
  while(($line = yield) != "END"){
    $logger->log($line);
  }
  $logger->close();
}
class Logger{
  public function close(){ echo "END;";}
  public function log($line){ echo $line; }
}

$gen = logger_generator();
$gen->send("First!");
$gen->send("Second!");
$gen->send("END");
```

Listing 3: Generator method used as a logger.

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. Moreover, it needs to be done the right way to ensure a correct order of execution.

Due to various design reasons, [Lippert, 2009b] C# also limits where *yields* can happen with regards to exception control blocks. They are not allowed in *catch*[3] and *finally* blocks[4] altogether and can only be in *try* blocks[5] 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, merely briefly supported [MDN] such behavior and as of ECMA 2015 has *yield* only as a statement.

That, nonetheless, does not mean that PHP is completely unique with regards to generators. In Python[6], another mainly interpreted dynamic language, *yields*

[3] [Lippert, 2009a]
[4] [Lippert, 2009b]
[5] [Lippert, 2009c]
[6] [docs.python.org]

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.

[7] [Lua.org]

2. .NET platform

The .NET platform, or any platform implementing the open CLI standard[1], stands on four pillars (Figure 2.1). The low level intermediate language CIL, the higher level languages such as C# and F# and their compilers to CIL, the base class library known as .NET framework, and - last but not least - the common language runtime, CLR, that actually executes the intermediate code.

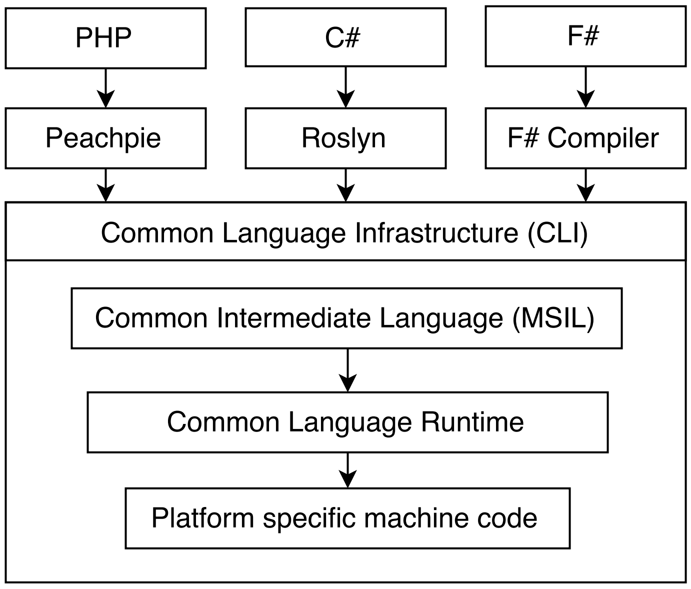

PHP
C#
F#
Peachpie
Roslyn
F# Compiler
Common Language Infrastructure (CLI)
Common Intermediate Language (MSIL)
Common Language Runtime
Platform specific machine code

Figure 2.1: Common language infrastructure.

We will talk about the C# and Visual Basic compiler Roslyn later, but neither the base class library nor the CLR will be covered extensively in this thesis. The common intermediate language, however, will be discussed in detail.

2.1 Common intermediate language

CIL is an assembler defined by the common language infrastructure [ECMA-335, 2012] to be a shared basis for all CLI languages (C#, F#, IronPython, ...) and runtime implementations (.NET, Mono, dotGNU, ...). It is platform independent and as such does not natively run on any CPU architecture. Instead, it must be either translated to the target platform's native code beforehand or - more commonly - executed by a virtual machine such as CLR.

Despite being an assembler, thus inherently low-level, CIL is actually object oriented and so has a deep understanding of reference types. Its instruction set reflects this with means to create new instances, access their members, and so on. The CLI specification also dictates that, by default, the CIL should be memory safe.

[1][ECMA-335, 2012]

2.1.1 Evaluation stack

CIL is a stack based assembler, therefore without the notion for registers. Instead, it defines a virtual evaluation stack. There are basically two types of instructions in CIL. Firstly, there are memory handling ones that either pop a value from the stack and store it in memory or load a value from memory and push it to the top. Secondly, there are instructions that actually do some processing. These pop a few values from the stack, process them in some way, and then store the result on the top of the stack.

There are a few important things to note about the evaluation stack [ECMA-335, 2012, Sec. I.12.4]. Firstly, all parameters and local variables actually live there. They are not ordinary stack values, though. Their place gets reserved and later cleaned automatically and they are not accessible through the normal push/pop instructions. Instead there are dedicated instructions to work with them.

Secondly, when exiting a function, the stack cannot contain anything but the returned value. Thirdly, there are instructions only to work with its top. There is no way to query all the elements in the stack, get its height, or to completely save or load it to/from memory.

Lastly, while not a rule, the stack is generally used as a store for temporal values instead of proper local variables. For example, an expression $2+3*5$ (Listing 4) would usually result in the load of constants 2, 3, and 5, a multiplication operation $(3*5)$ (see *IL_0009*), at which point the stack would contain 2 and 15, and finally a plus operation (see *IL_000a*) that would leave the stack with 17 at its top.

All of these mean that you cannot simply pause and save the execution of a method at an arbitrary point with just one or even a few CIL instructions. To completely capture the current state, you not only need to save all the local variables and parameters somewhere off the stack, but you must also do the same for every temporal value that might at that moment live on the stack. And there is no simple way to query what is there. You either need to construct the information in some other way or restrain yourself to saving the state only when the stack is empty.

2.1.2 Exception handling

The last notable thing about CIL is that it has a notion of exceptions and their handling blocks. Try, catch, and finally are all first class citizens in the language and are bound by a number of rules [ECMA-335, 2012, Sec. I.12.4].

CIL does not permit jumping / branching into any exception handling block[2] unless the source of the jump / branch is within the same block. You can only enter catch and finally regions through the proper exception handling mechanism. And lastly, to leave any of them[3], you need to do it via a designed instruction that, in case of try and catch blocks, ensures any potential finally region gets run. Therefore, you can neither jump in the middle of a try block nor execute a catch / finally block without throwing a proper exception first.

[2][ECMA-335, 2012, Sec. I.12.4.2.8.2.7]
[3][ECMA-335, 2012, Sec. I.12.4.2.8.2.8]

```
public void M(int a) {
  int b = 3;     // IL_0001 & IL_0002
  int c = 5;     // IL_0003 & IL_0004
  G(a + b * c);  // IL_0005 - IL_000b
}
public int G(int a){/*Something*/}

.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 value from stack to local variable 0
  IL_0003: ldc.i4.5  // Push 5 onto the stack as int32
  IL_0004: stloc.1   // Pop value from stack to 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 by G) from the stack
  IL_0011: ret       // Return from method, possibly with a value
} // end of method C::M
```

Listing 4: Simple method in C# and CIL.

3. Peachpie project

The Peachpie project[1] aims to create a bridge between PHP and the .NET ecosystem. While its development started in early 2016 it builds upon the foundations of a much older project, Phalanger[2], first released in 2004 and also originally developed at the Charles University.

While both projects share the same end goal - bring PHP to .NET, their implementation is quite different. Phalanger, due to being first released before even .NET 2.0 shipped, had to implement almost everything on its own. Peachpie, on the other hand, relies heavily on components provided by the Roslyn infrastructure in the compiler and by the DLR at runtime.

Also, while Phalanger supports PHP 5.4 as the highest version, Peachpie was built for PHP 7.1 and beyond from the very beginning. The last major difference is that Peachpie, unlike Phalanger, runs not only on full .NET and Mono but also on the multi platform .NET Core framework.

The Peachpie project is, as of early 2017, still in an active pre-version 1.0 development by the open source community. As such, its architecture is not yet finalized and might change in the future, potentially rendering the following chapter inaccurate.

3.1 Peachpie architecture

Peachpie consists of three more or less separate parts. A compiler that takes PHP sources and produces .NET assemblies, a runtime library that provides support for various dynamic features of PHP, and a reimplementation of the PHP base class library with its most popular extensions.

Due to the topic of this thesis being an implementation of a compiler feature, we will focus mainly on the compiler and only briefly discuss the runtime library. The base class library, while interesting, is completely irrelevant for our work and will be left out.

3.2 Peachpie compiler

The compiler itself is built on the architecture of an open source C# and Visual Basic compiler platform entitled Roslyn. The compilation is logically divided into four main phases (Figure 3.1). First a parser takes a PHP source and creates an abstract syntax tree. In Peachpie, this step is actually offloaded to a third party open source PHP parser[3]. Then, Peachpie takes over and binds the AST to a semantic representation in the form of a control graph, essentially creating an abstract form of the final program. The binding phase is also responsible for lowering higher level language constructs.

Next, the semantic graph is used for an extensive data-flow analysis with the intention to resolve dynamic types and generally eliminate as much dynamic

[1]peachpie.io/
[2]github.com/DEVSENSE/Phalanger
[3]github.com/DEVSENSE/Parsers

behavior as possible. This step is important mainly for performance reasons. Dynamic dispatch and access at runtime inherently causes a performance hit, especially on the .NET CLR that, despite being language agnostic, is still tuned mostly for C# and VB, both of which are statically typed languages.

In the last phase, the semantic graph is used to emit the final CIL code and produce a complete .NET assembly. While Peachpie controls the emit of each individual CIL instruction, their specific bytecode realization, possible CIL level optimizations, and the assembly structure creation are all handled by the Roslyn components *ILBuilder* and *PEBuilder* respectively.

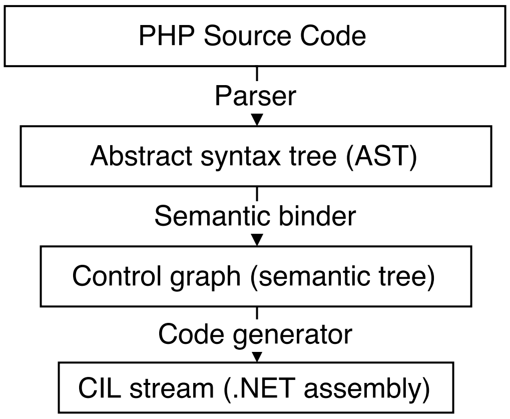

PHP Source Code
Parser
Abstract syntax tree (AST)
Semantic binder
Control graph (semantic tree)
Code generator
CIL stream (.NET assembly)

Figure 3.1: Peachpie architecture.

There is, of course, more to the compiler part. It also includes a number of code analyzers to catch common PHP bugs, provides an extensive API surface to support projects such as PHP snippets or a Visual Studio Code extension, and much more.

3.3 Semantic graph

Since our approach to implementing a generators support is based on a semantic graph transformation, let us explore it a bit more. Unlike the AST, which is a structured representation of the source code, the semantic graph corresponds more closely to an abstraction of the final program.

It knows the types of all expressions, has all method calls as well as variable/field accesses resolved and bound to specific semantic symbols, and generally contains all the information needed for a future compilation.

3.3.1 Statements and expressions

Before going into specific details about the semantic graph itself, let us properly define the difference between expressions and statements first. An expression is a combination of values and operations that produces a new value while potentially also having side effects.

For example, plus is a binary expression that creates a new value from its two children expressions. Method call, if the method returns some value, is an expression as well. Statement, on the other hand, is an operation that only has some side effects and does not carry a value itself. A good example of a statement is goto jump.

This means that in expression trees (Figure 3.2), a computation abstraction where each node is an operation that takes the values of its children and produces a new value, statements can only be at the top. Since they do not have a value of their own that could be consumed by their potential parent, they simply cannot have a parent node.

It is also good to note that while it is simple to transform an expression into a statement from - you simply throw away its value, you cannot do it the other way and use a statement in places where an expression is expected.

3.3.2 Graph structure

With that out of the way, the semantic graph is fundamentally a forest in which every method declared in the source code or synthesized by the compiler corresponds to its control flow graph. Each graph consists of two types of elements: edges, representing control flow constructs such as loops, branches, and exception handling constructs, and blocks, simple containers holding standalone semantic statements like method calls, assignments, variable declarations, and so on.

These individual statements, can be in the form of rather complex graphs themselves (Figure 3.2). They can be arbitrary expression trees with a statement at the top. For example, an assignment statement has two expression children, the variable and the one representing the value being assigned. The value expression can also be arbitrary and have children of its own and so on. Semantic expressions and statements are called bound within Peachpie.

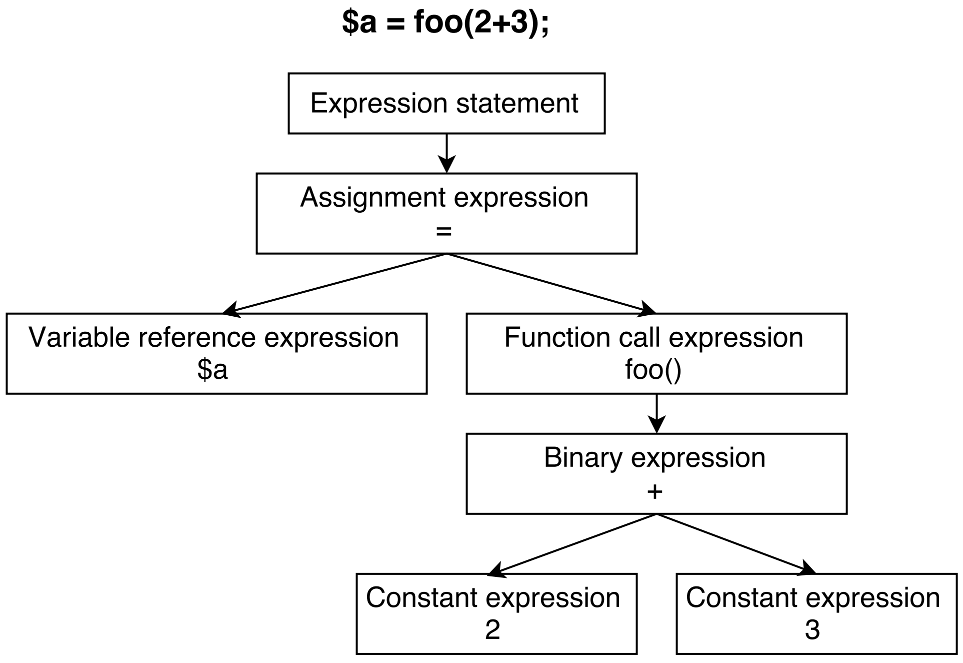


Figure 3.2: Expression tree.

The control flow edges do not have to be simple and connect only two blocks either. In fact most edges connect multiple blocks and some even have references to individual expressions (Figure 3.3). For example, a switch edge connects a source block, a switch variable expression, an arbitrary number of case blocks with their case value expressions, and an end block. Other edges are implemented similarly.

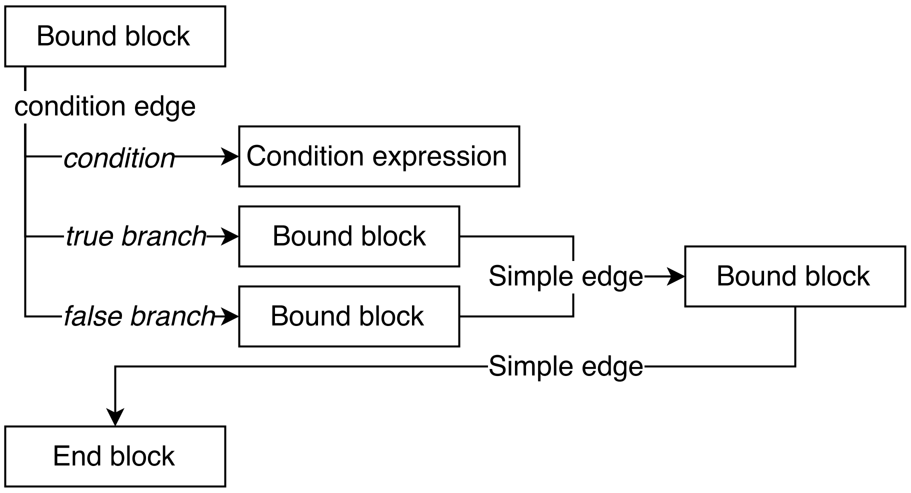


Figure 3.3: Condition edge.

The last type of objects to note with regards to the semantic graph are symbols. They represent declared objects. These include types, namespaces, methods, fields, variables, and parameters. Read and write accesses to fields, variables, and parameters inside methods are still represented as bound expressions, however. They are bound expressions that hold a reference to these symbols and use them as an identifier of the actual place where their value is.

3.3.3 Graph creation

There are two components in Peachpie compiler responsible for the individual method's control flow graph creation: *BuilderVisitor* and *SemanticBinder*. *BuilderVisitor* is a higher level component traversing the top level of a method's abstract syntax tree and creating the aforementioned control flow edges and bound blocks in the process.

It uses the *SemanticBinder* to fill these bound blocks with bound statements and to create sporadic bound expressions needed for the edges, e.g. a switch value expression for a switch edge. Specifically, it goes through statements and control flow construct in the method's AST and does two things. It either adds a newly bound statement into the current bound block or, on control flow constructs, creates new bound blocks. When doing so, it fills them with statements, connects them to the previous current block, and sets the last of them as the new current block.

The *SemanticBinder* is a component that takes a statement or an expression in the form of an abstract syntax tree and creates its semantic representation, either a bound statement or a bound expression, that can be used in the resulting semantic graph.

It also handles the full complexity of the statements/expressions. When it gets asked by the *BuilderVisitor* to bind an assignment, it creates not only the bound assignment statement itself, but it binds its children, and their children, and so on as well, returning the full bound subgraph.

That is the reason why the *BuilderVisitor* only goes through top level statements and does not care about individual expressions. With the exception of those within edges, as noted, each expression is part of a larger expression tree under some statement that gets bind as a whole tree by the *SemanticBinder*.

3.4 CIL emit phase

In Peachpie, the CIL code generation is based solely on the semantic graph. Each of its elements, be it an edge, a statement, or a symbol, has a method, *Generate* or *Emit*, that can create the element's complete CIL code representation. These methods do not produce the bytecode themselves. Instead, they use a component called *CodeGenerator*, a thin wrapper around Roslyn's *ILBuilder*.

3.4.1 Code generator

CodeGenerator is fundamentally an abstraction of a CIL code stream. It can do two things: append either individual CIL instructions or their short sequences to the current code stream and realize the stream into actual bytecode. The only higher level service it provides is the ability to change where it should look for certain important items.

One can, for example, set an arbitrary variable as a method's *this* object or specify that local variables should live in some PHP array instead of on the evaluation stack. Appending a load from a local variable then results in the correct CIL sequence being emitted. That means either a load from the locals part of the evaluation stack or, when the place of locals was changed on the current *CodeGenerator*, some *PhpArray*, which itself might need to get loaded from some field first.

This is used, among other places, when there are indirect variable accesses in a method[4]. Because of their indirect and thus dynamic nature, it is impossible to resolve and bind them at compilation time. In addition, there are no CIL instructions to create a new local variable nor to access a variable by its name. Afterall, local variables in CIL are just slots in the evaluation stack. Therefore, for methods with indirect variables, its locals must be moved from the evaluation stack to a *PhpArray* that supports both of these operations.

3.4.2 Emit

The act of emit is very similar to the binding phase. When *Generate* is called on a semantic item, it emits CIL representation of not only the item itself but also of all that is under it in the semantic graph. A code generation for a method symbol (Figure 3.4), for example, causes an emit of all of its bound blocks and edges, each of which triggers the emit of their bound expressions and statements, effectively ending up generating CIL for the whole method's body.

[4]php.net/manual/en/language.variables.variable.php

As such, the emit is effectively a mixture of pre and post-order traversal of the semantic tree. First, the emit of the current item starts, subsequently its children from left to right get fully emitted, and then it finishes. Due to that and the fact that execution follows the emitted CIL code, there is an invariant. The left children represent code that is executed before the right children and nodes lower in the tree finish before their parents.

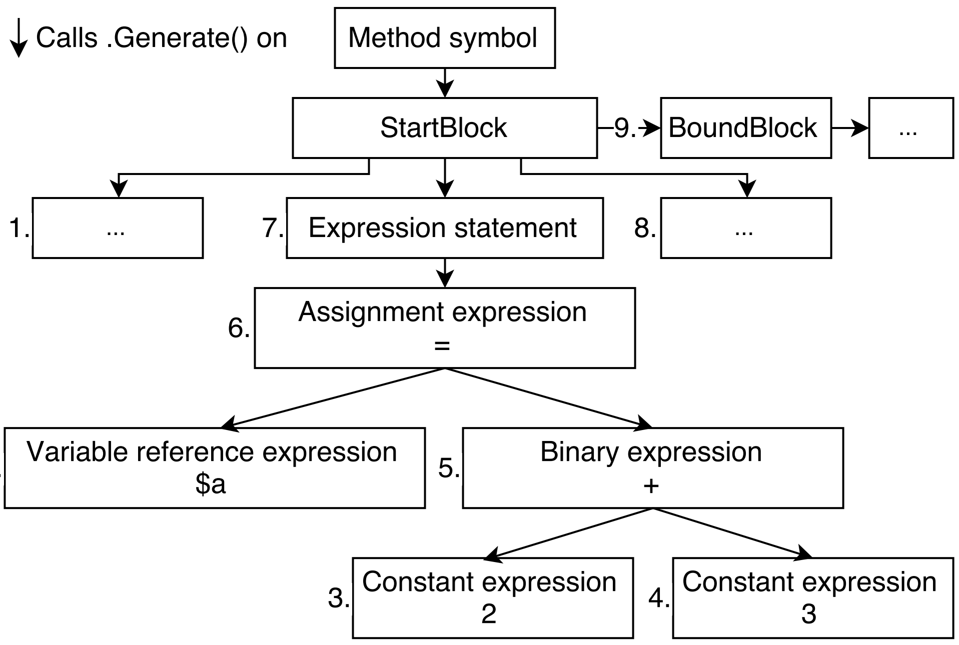


Figure 3.4: Emit order of a method.

The individual *Generate* or *Emit* methods are completely independent. They only append CIL instructions representing their semantic node to the *CodeGenerator* instance they got passed as a parameter. In case their respective nodes have any children within the semantic graph, they also call *Generate* on them, always passing the *CodeGenerator* instance they themselves got. This ensures that in the end, the one *CodeGenerator* instance contains CIL code for all the method's statements, expressions, and edges.

3.4.3 Generate methods' invariants

While individual nodes can emit themselves however they want, there are two rules that must hold true. The code emitted by a bound expression must load and leave its value on the top of the evaluation stack. Other than that, it cannot leave anything else there nor can it remove something. Their *Generate* methods also have to return a symbol representing the expression's IL type.

For bound blocks, statements, and edges, the rule is similar with the difference that, since they do not have an inherent value, they cannot leave anything on the evaluation stack at all. Their methods do not return anything.

These rules ensure a number of things. First, all expressions are basically interchangeable. An emit of a binary expression does not need to care about the operands' types. It can simply call *Generate* on them and know that the evaluation stack will contain their values, independent on whether they are constant expressions, method calls, or something else.

Also, since neither statements nor edges can leave anything on the evaluation stack and statements can only be at the top level of the semantic tree, the evaluation stack is guaranteed to be empty in the beginning of each bound statement's execution. This is important because there are statements, such as return or goto, that transfer execution and therefore need the evaluation stack to be empty.

3.5 Peachpie runtime library

The runtime library consists of a number of important types needed to support the dynamic nature of PHP. Probably the most important one is *PhpValue*, a managed counterpart of the Zend Engine's *ZVal*[5]. It is a type used everywhere the data-flow analysis can not ensure a more specific type. It is a lightweight structure that can hold a primitive value (number, string, or boolean), *PhpArray* (essentially a hashtable), reference to a proper class instance, or - in case of a reference variable - a link to another *PhpValue*. It also supports all the required type conversions and serves as a true representation for an arbitrary PHP value.

Another core type present in the runtime library is *Context*. As suggested by its name, it holds the context of the current execution. It contains defined global methods, variables, and constants, the value of static fields, and much more. It is a rather special type because it gets silently passed to every PHP method as their first argument throughout the whole execution.

When a library originally written in PHP is used from another .NET language (C#, F#, . . .), the context has to be explicitly created and passed into it as part of the method's invocation. If the original PHP app is used on its own, the context gets created automatically in the beginning before any user code is run.

Other than that, the runtime library also contains a function call and variable access resolution logic needed to support dynamic behaviors at runtime, a variety of types needed for certain PHP features, such as base class for lambdas, and a full reflection support.

[5]php.net/manual/en/internals2.variables.intro.php

4. Generators in other platforms

In this chapter, we will take look at two very different approaches towards supporting generators. First, we will describe the way they are implemented by the C# compiler Roslyn. The combination of Roslyn and C# was chosen because, despite the differences between generators in C# and PHP, it is still very similar to our Peachpie and PHP mix.

Both Roslyn and Peachpie compile their respective languages into the CIL, Peachpie's compiler is de facto based on Roslyn's architecture, and generators in PHP are fundamentally a superset of what they are in C#.

IronPython and its compiler based on a DLR was also considered, mainly due to Python's generators being closer to PHP's. The similarity of compiler platforms and the fact that IronPython offloads a lot of details to the DLR, which is by design generic and therefore needlessly complicated for our use, prevailed, however.

The second implementation we will talk about in this chapter is Zend Engine's for generators in PHP. In spite of the fact that we cannot use it as an inspiration, because it relies on a native support provided by the runtime, it is useful to mention it at least briefly. Afterall, it is the implementation we are trying to mimic.

4.1 C# and Roslyn

As said previously, the CIL, into which .NET implementation of C# gets compiled, does not have a native understanding of generators. There is, for example, no instruction to yield or to automatically construct an iterator type instance with all the appropriate methods. Therefore, the Roslyn compiler has to support generators by lowering them, in essence transforming them into lower level language constructs.

There are two main components responsible for that. First, there is a rewriter that takes the original generator method and transforms it into a normal method that only uses constructs the CIL supports. We will come back to it later. Then, there is the type implementing the *IEnumerator* interface (Figure 1.1), whose instance gets returned from the generator method and which actually represents the generator.

4.1.1 Iterator object and generator methods

There is no single type implementing the *IEnumerator* interface. For each generator method, the Roslyn compiler synthesizes one separate iterator type. While most of their *IEnumerator* methods are simple, and actually the same for all of them, there is one that is always unique - the *next* method. This one actually contains the implementation of the original generator method, only now transformed and turned into a state machine by the rewriter.

The actual original generator method has its body replaced with compiler generated code that instantiates, initiates, and returns the corresponding synthesized type (Figure 4.1). Therefore, it is a normal CIL method that returns an iterator containing a transformed version of the method's original body as its *next* method.

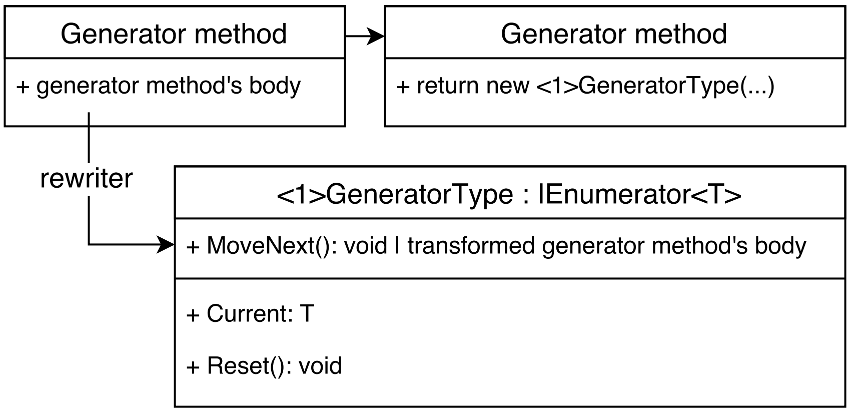

Generator method
+ generator method's body
Generator method
+ return new <1>GeneratorType(...)
rewriter
<1>GeneratorType : IEnumerator<T>
+ MoveNext(): void | transformed generator method's body
+ Current: T
+ Reset(): void

Figure 4.1: Generator method, generator's next method, and the generator type.

4.1.2 Rewriter

The rewriter has to take care of three things while transforming the original generator method's body into the iterator's *next* method. It has fix the references, that expected the method to be where it was instead of on the generated type, and handle both the state saving on yields and the state retrieval in the beginning of the method.

The only references that care where the method actually is are to *this*. Fortunately, the *this* instance can simply be captured in the original generator method, passed to the generator during its initialization phase, and kept there in a field. All references to the original *this* variable can then be rewritten to references to the generator's field holding the captured *this* instance.

As for state saving, due to the fact that yield is only a statement in C# and an invariant in Roslyn, that the evaluation stack is always empty in between separate statements, there are guaranteed to be no temporal values on the stack before any yield gets executed. This means that the only state that needs to be saved are local variables, parameters, and the position, in essence the next statement to be executed.

4.1.3 Local variables parameters

To handle the first two thirds of the state, the rewriter creates a new field on the corresponding iterator type for each local variable and parameter. Then, it replaces all, both read and write, references to these original local variables and parameters with references to the newly created fields (Figure 4.2). The result is that there are no accesses to local variables or parameters in the rewritten method. Also, all values now live on the iterator instance which means they are persistent in between individual calls to the *next* method.

The situation regarding parameters is, in fact, a bit more complicated. As defined by the *IEnumerator* interface, the *next* method cannot have any parameters.

Fortunately, there are no references to the parameters inside the *next* method after the rewrite, only to the instance fields they got replaced with. The fields still need to be initialized with their values, however.

That can be done similarly as the this reference was handled in the original generator method. The method has access to both the original parameters and the iterator instance before any code from the *next* method has any chance to access the fields. Hence, after the iterator instance is created, the parameter values get assigned to their corresponding fields as part of an iterator's initialization phase.

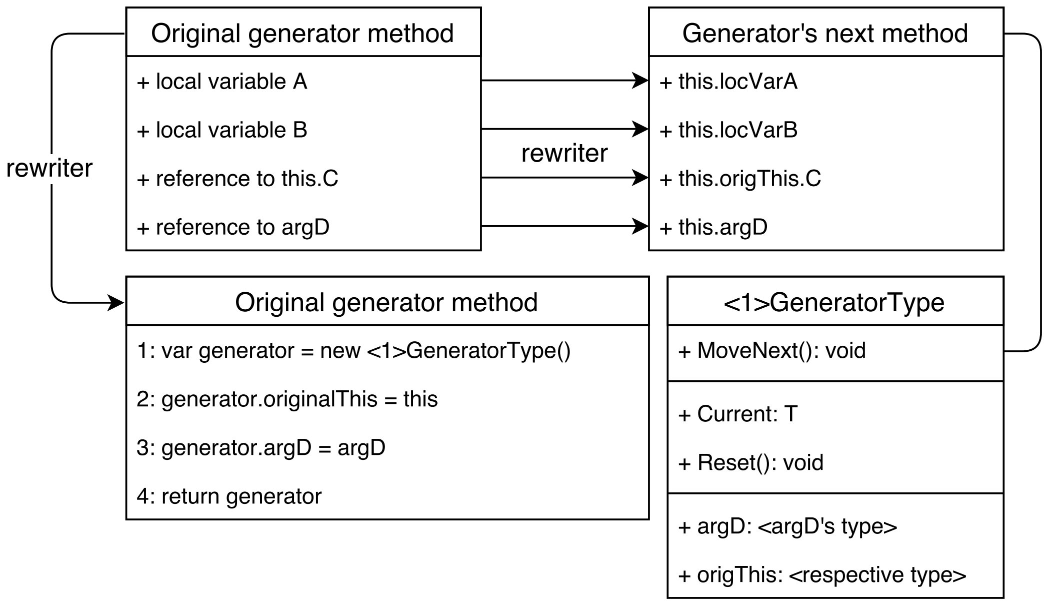


Figure 4.2: Local variables rewrite for generator methods.

4.1.4 Execution position

The last thing that needs to be taken care of is saving the point where the last yield happened, effectively the place a subsequent call of the *next* method should continue from. To handle that, the rewriter does two main things. It replaces each *yield* with a number of statements and creates a jump table in the beginning of the *next* method (Figure 4.3). A new field called state, representing which yield the generator exited with last time, is also created on the iterator instance.

Each *yield* gets lowered into four separate statements. First, an assignment of the yielded value to a current field on the iterator instance. This field is used by the *IEnumerator*'s *current* method as its backing field. Second, an assignment of the current *yield*'s index to a state field on the iterator instance. The order of these indexes can be arbitrary, only a uniqueness among other *yield*s' indexes within the same method is required. Third, a normal *return* from the *next* method. And finally, a *yield*'s label based on its index that can be used as a target for jumps.

The jump table in the beginning of the *next* method is fundamentally a switch statement that, based on the current value of the iterator's state field, jumps to a corresponding *yield*'s label. Within the four statements created by rewriting one yield, the state field assignment and the label are connected. The assignment sets a state value whose corresponding case in the switch table contains a jump to the label. And since the indexes and therefore states are unique, it is guaranteed that this always holds true.

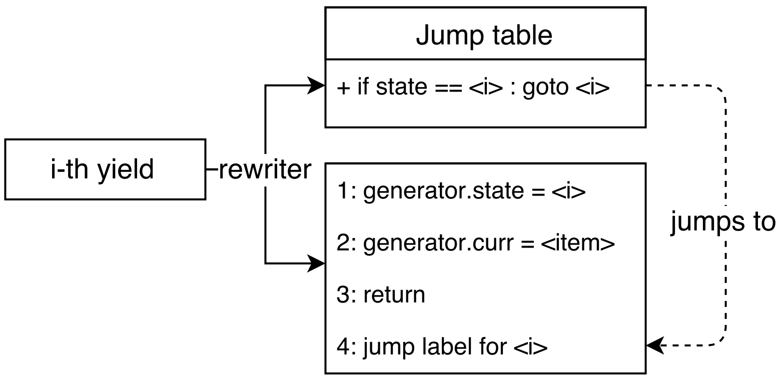


Figure 4.3: Local variables rewrite for generator methods.

That way when the *next* method is called repeatedly, it always resumes with the statement that is directly after the *return* the method exited with before. This happens simply due to the fact that whenever you hit a *yield*, and therefore a *return*, two things are true.

First, just before returning from the *next* method, an index of the hit *yield* gets assigned to the state field. Second, the state field does not change unless the *next* method is called again. And when that happens, the switch table in the beginning of the *next* method jumps to the label that was created by rewriting the same *yield* as the last evaluated state assignment. Which is, as we have already proven, the label directly after the *return* the method exited with previously.

```
// Original method's body
Yield 5;
Yield 10;

// New method's body
switch(this.state){  // Jump table.
  case 1:            // If state represents 1st yield.
    goto: Label_1;   // Jump to label for 1st yield.
    break;
  case 2:            // If state represents 2nd yield.
    goto: Label_2;   // Jump to label for 2nd yield.
    break;
}

this.current = 5;    // Set yielded item as current.
this.state = 1;      // Set state for 2nd yield.
return;
Label_1;             // Label for 1st yield.

this.current = 10;   // Set yielded item as current.
this.state = 2;      // Set state for 2nd yield.
return;
Label_2;             // Label for 2nd yield.
```

Listing 5: Original and rewritten generator method's body.

Naturally, this is not all the Roslyn compiler does to support generators in C#. It is, however, more than enough for us to design our own implementation in the Peachpie compiler.

4.2 PHP and Zend Engine

Unlike CIL and CLR respectively, the reference PHP runtime Zend Engine understands generators natively [Popov, 2017]. As such, it is able to execute yields without having to lower them into simpler PHP constructs.

Not going into details and slightly simplifying, the execution state of the Zend engine is represented by a virtual machine stack. This stack contains individual stack frames, each corresponding to a method's execution. When a method is called, a new stack frame gets created, initiated, and pushed on top of the stack. When the method returns, its stack frame gets popped.

Each frame contains the complete information about a method's execution state, all of its local and temporal variables, arguments, the returned value, and an index of the last executed statement, to name a few. Therefore, if one needed to save the execution state of a method, storing its frame stack would be enough.

And that is exactly what the Zend engine does when it encounters a yield expression. It creates a new generator object, copies the current stack frame into it, performs a number of other tasks, such as setting its current key and value, and finally returns the iterator. In this context, the current stack frame is the one representing the execution state of the method with yields and therefore, in essence, the generator method.

Later, when the *next* method is called on the returned generator instance, it restores the stack frame previously saved on the generator object to the top of the virtual machine stack and resumes the execution. This effectively causes the generator method's execution to continue from the very point where the last yield was encountered and thus where it stopped. On subsequent yields, the runtime sets the generator's fields such as key and current, updates its saved stack frame representing the generator method's current state, and returns.

The description above is a simplification of the actual process that happens in the Zend engine, with details regarding yields in exception blocks and inside function calls completely omitted. However, it still provides a good high level overview of how generators are implemented in PHP's reference runtime and how it is different to Roslyn's approach.

5. Generators in Peachpie

The goal of this thesis is to enable Peachpie compiler to handle PHP generator methods while keeping as much of their original semantics as possible. That means we do not want to change their behavior and want to enable all the features they offer in PHP, only now compiled to CIL and executed either by the CLR or another CLI environment.

As noted in previous chapters[1], this in itself is complicated, because unlike the PHP runtime Zend Engine, the CIL and CLR do not have a native support for generators or generally pausing the execution of a method at arbitrary points. Also, almost all other CIL based languages with generators, such as C# or F#, that implement them by compiler transformations have them in a substantially more limited form than PHP.

Other than that, we also want to reuse existing Peachpie infrastructure and only implement generator specific bits when necessary. While this goal is not as important for our immediate work, it is necessary for the project as a whole. Cluttering the compiler with logic for a feature that is not actually used as often would simply be inexcusable.

5.1 Basic generators implementation

Before dealing with all the complexities of PHP's generators, let us first explore how an implementation of their limited subset would work within Peachpie. Specifically, we will ignore *yield* in exception handling blocks and expect it to be only in places where it could happen as a statement, i.e. no *yield* inside an expression tree, for this chapter.

Much like Roslyn's approach, our implementation of generators within Peachpie will also be based on transforming the original generator method into an iterator's *next* method. So as not to repeat ourselves, we will only point out the differences in the next section.

5.1.1 Iterator object

Unlike in C#, where generator methods are free to return any object implementing an *IEnumerator* interface, the PHP specification dictates that the returned object must be an instance of a *Generator* type [PHP.Net, a, Popov, 2012]. This means that in Peachpie we cannot just synthesize a new type for each generator method as Roslyn does.

If we were to do that, all reflection methods and type checks would report the actual synthesized type on the returned iterator instance instead of the *Generator* type, as required by PHP's specification . We could theoretically hard code exceptions for these synthesized types into all methods that query an instance's type, but that would go against our goal to implement as little feature specific code as possible.

Instead, we must create one generator type in Peachpie's runtime library and use it as a basis for all generator methods to return. That approach, however,

[1]Chapter 4.1

carries some limitations with it. The generator type can now include only shared code and fields. That means neither a specific *next* method's implementation nor fields for lifted local variables from said method.

Other than that, the *Generator* type can be practically the same as the ones synthesized by Roslyn as it is a simple implementation of PHP's *Iterator* interface (Listing 6). It can hold a captured reference to the *this* instance of the original generator method, a state field to know what point the *next* method should continue from, fields for the *current* element and, since we are in PHP now, its *key*.

```
public delegate void GeneratorStateMachineDelegate(
  Context ctx, object @this, PhpArray locals,
  Generator gen);
public class Generator : Iterator
{
  readonly Context _ctx;
  readonly GeneratorStateMachineDelegate _stateMachineMethod;
  readonly object _this;
  readonly PhpArray _locals;
  internal int _state = 0;
  internal PhpValue _currValue, _currKey;
  public void next() =>
  _stateMachineMethod.Invoke(_ctx, _this, _locals, gen: this);
}
```

Listing 6: Simplified version of the Generator type.

5.1.2 Next method implementation and local variables

The *next* method's implementation problem is easily solvable. The shared generator type can hold a delegate to an implementation of the *next* method instead of the method itself. This enables Peachpie compiler to synthesize the *next* method anywhere and then to assign its delegate to the generator. The generator must still implement some *next* method to comply with the Iterator interface but it can be a shim that only calls the saved delegate.

There is only one restriction with regards to the actual *next* method's placement. The transformed method must be accessible from within the original generator method. The reason is that the original method is where the instantiation and initialization of the generator, thus also the creation and assigning of the delegate, happens.

One such suitable place is the enclosing type of the original method, where it can always be synthesized as a static method. It being a static method is not a problem because, as mentioned in the chapter about Roslyn's implementation, a reference to the enclosing type's instance is passed as a *this* parameter. And since the enclosing type could be a static class, it cannot be a normal instance method anyway.

The inability to add fields to the generator type can also be overcome. As described in the CIL emit phase chapter[2], Peachie's *CodeGenerator* supports specifying where local variables should live within a method with the option to, for example, move them to a *PhpArray*.

With that, a *PhpArray* field can be added to the generator type and we can specify that all of the *next* method's local variables should live on it. Because parameters are considered local variables in Peachpie, this approach handles them as well. They only need to be initialized with their values in the original generator method. That way, the *next* method's local variables and parameters get lifted to the generator type the same way as in Roslyn, with the only difference being that they do not get lifted to individual fields but to a single *PhpArray* (Figure 5.1).

5.1.3 Accessibility of fields on the Generator type

Moving the *next* method outside of the generator type means that the method cannot access its fields such as *current* or state directly through a *this* reference. That is a problem, because the method needs to both read and write these fields to progress the generator. An effective solution is to pass the generator instance as a parameter through the *next* method delegate - in essence to not only call the delegate from within the generator's own *next* method, but to call it with a *this* reference as a parameter.

That on its own would be enough if all the fields on the generator type were public. That is, however, not our objective. We want the generator type to have the same public API as it does in PHP and there are no such public fields in PHP's *Generator*. Therefore, we need to find a way to access the fields from a method within the user's assembly, the transformed *next* method the generator has a delegate to, without having to make the fields accessible to other user code.

One way to do this is to make the generator fields internal and create public getter and setter methods for these fields in the Runtime library. Since the generator type also lives there, the methods can access its fields and, because they are public, they can be used from within the synthesized *next* method (Figure 5.1). The methods can be simple static getters and setters, always taking a generator instance as a first parameter and either returning an appropriate field's value or taking its value as a second parameter and then setting it on the instance.

While it is true that this approach still opens a way for the user to modify the generator's internal state, it has to be done through special methods from Peachpie runtime library and as such, it can hardly be done by accident. It also ensures a compliant public API of the *Generator* type.

5.1.4 Context handling

Being in PHP, we need to ensure that the correct PHP *Context* gets passed to our moved *next* method. There are two ways to do it. Current *context* can either be passed as part of each call to the generator methods, and subsequently via a delegate to the *next* method, or it can be captured once during the generator's initialization and then reused the same way as is the *this* instance.

Neither approach is inherently wrong. Passing the *context* with each call ensures the current one is used even in situations where one generator instance

[2]Chapter 3.4.1

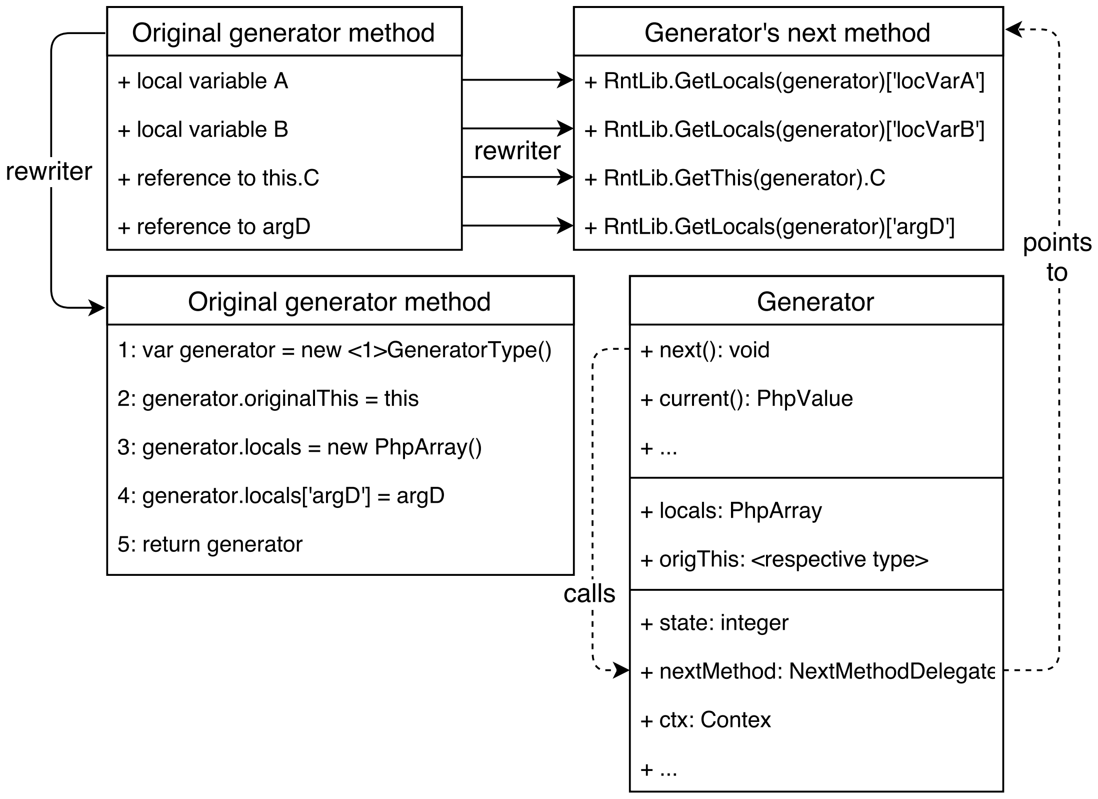

Original generator method
+ local variable A
+ local variable B
+ reference to this.C
+ reference to argD
rewriter
rewriter
Generator's next method
+ RntLib.GetLocals(generator)['locVarA']
+ RntLib.GetLocals(generator)['locVarB']
+ RntLib.GetThis(generator).C
+ RntLib.GetLocals(generator)['argD']
points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

Figure 5.1: Transformation of the generator method.

is used with multiple PHP *context*s. That can happen, for example, when PHP code is called from some other .NET language and multiple *context*s are created manually. This approach is also more in line with how *context* on normal instances is handled. It is not captured in the constructor and then used by all the instance methods, but always passed as a parameter.

On the other hand, capturing the *context* on the generator's creation better represents the idea that the generator is a fully self-contained object. It is also marginally easier to implement and provides better opportunities for interop between PHP generators and other .NET languages. This way, a generator can be created in PHP and then used elsewhere as a normal iterator, without having to explicitly keep and supply its *context*. Thus, the capture once in the original generator method approach was chosen.

5.1.5 Rewriter

Due to architectural differences, we will not have a standalone rewriter component in Peachpie. While it would be possible, there are, as of writing this thesis, no other candidates that could make use of them within the compiler. And adding a generic support just to have one rewriter for generators goes against our goal to keep the implementation as simple as possible. Instead, our implementation will rely on support by the *SemanticBinder*, slightly changed emit of a *MethodSymbol* and *StartBlock*[3], and a new semantic node.

[3]Chapter 5.1.7

As long as we limit ourselves to *yield* only at places where it could be as a statement, which is the temporal restriction we have set for this chapter, the support provided by the *SemanticBinder* can be minimal. It needs to do two things: bind the new semantic object - *BoundYieldExpression* - when it encounters the AST's *YieldExpression* and mark the method's symbol as a generator.

5.1.6 Bound yield expression

The *BoundYieldExpression* can be a rather simple semantic node with two children: the yielded key and yielded value expressions. It should generate CIL to set the yielded key and value fields on the generator instance, update its state, *return*, and mark a label for the subsequent continuation (Figure 5.2).

Due to being an expression, albeit for this chapter limited to places where it could also be a statement, it needs to push and leave its value on the evaluation stack. However, since its value will not be needed due to our restriction, it can just as well be an empty *PhpValue*. The value will always get discarded, anyway. The restriction also handles the problem that we are emitting a *return* from within an expression, i.e. in a situation in which the evaluation stack might not be empty.

It is true that all of the *BoundYieldExpression* could be replaced with a number of normal PHP statements by lowering. That would, however, require the *SemanticBinder* to be able to produce multiple semantic statements for only one AST node, and for the *BuilderVisitor* to accept them. And while such support could be added, it was decided that it would be too complex.

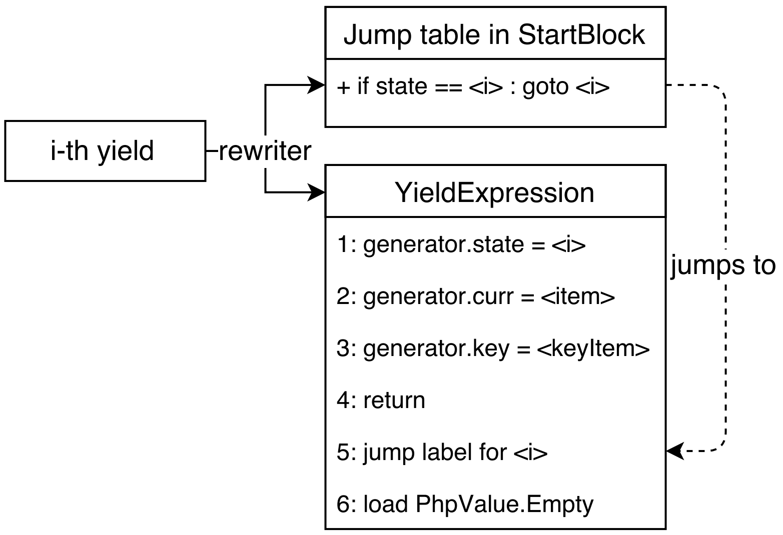


Figure 5.2: Rewrite of a yield expression.

5.1.7 Start block

The *StartBlock* is a special instance of a *BoundBlock* that is present in the beginning of each method's control flow graph (Figure 3.4). As such, its emit routine is the perfect place to generate a jump table for generator methods. It can, when its enclosing method symbol is a generator, query all present *yield* statements that were previously set by, for example, the semantic binder and generate the whole jump table.

5.1.8 Method symbol

The method symbol's emit must be changed as well. When it represents a generator, it cannot simply generate the CIL representation of its body. If it did that, it would not produce a method that returns an iterator as it should, but a method that implements the iterator's *next* method.

Instead, three things need to happen. First, a new static method representing the generator's *next* method must be synthesized in the enclosing type. Second, the original body needs to be emitted inside the synthesized method with its *CodeGenerator* set to offload local variables into a generator's locals field. As explained earlier, the synthesized *next* method accepts a *Generator* as a parameter.

Third, a sequence of statements that create, initiate, and *return* a *Generator* instance must be emitted as the actual current method symbol's body, producing a method that returns an iterator. As part of the initiation phase a delegate to the synthesized *next* method must get created and assigned to the newly constructed generator instance. Also, values of all parameters need to get copied to the generator's locals array, as previously discussed.

5.2 Yield as an expression - theory

With that, we have described a design of a generator's compilation within the Peachpie platform with a featureset limited to more or less C# generators. Now, let us broaden it with the support for *yield* as an expression. Before going into details on the specific implementation, let us first take a look at the general idea behind our approach.

As said before, a *yield* being an expression is a problem, because an expression can happen in a situation where the CIL evaluation stack might not be empty. Since *yield*s include a *return* and returning with a non-empty evaluation stack is forbidden, it does not go well together. Even if it were allowed, there would still be the problem that the non-empty evaluation stack would represent some sort of state - one that would need to get saved and then retrieved upon the continuation.

5.2.1 Possible approaches

Fundamentally, there are two possible ways to approach this problem. One can either come up with a mechanism to save and then retrieve the evaluation stack or rearrange the semantic graph so that *yield*s are only in places where they could happen as statements.

While the first approach might be appealing, after all it more closely mimics the Zend Engine's way of handling yields[4], it is almost impossible to implement. Because the CIL does not have any instructions to query the contents or to completely save/load the evaluation stack, the compiler would have to do it manually.

[4]Chapter 4.2

That means it would have to track the stack's content throughout the compilation and then emit individual instructions to save/load its content, one element at a time.

That would mean two things. First, we would either have to create our own version of the *CodeGenerator* that would be able to keep track of what the evaluation stack contains at any moment or we would have to change the emit of each semantic node to save the information about what it puts on the stack explicitly. Both of these would be relatively complex to do and, in case of the second approach, even to maintain due to possible new semantic nodes. Second, either of them would mean an increase in memory usage because we would need to remember information previously not required, all of which just to support only a *yield* as an expression.

On the other hand, the second approach, to rearrange the semantic tree, requires only a few local implementation changes and does not cause a substantial memory consumption increase. Essentially, it is based on the idea that we can break an expression tree into a series of statements while keeping the meaning and order of execution the same.

5.2.2 Branch capture & yield splitting

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, acting as the equivalent of a C# *yield* statement. The other node is an expression that represents the sent value. If the expression directly follows the statement, the result is, in terms of emitted CIL, the same as with one combined *yield expression* (Figure 5.3).

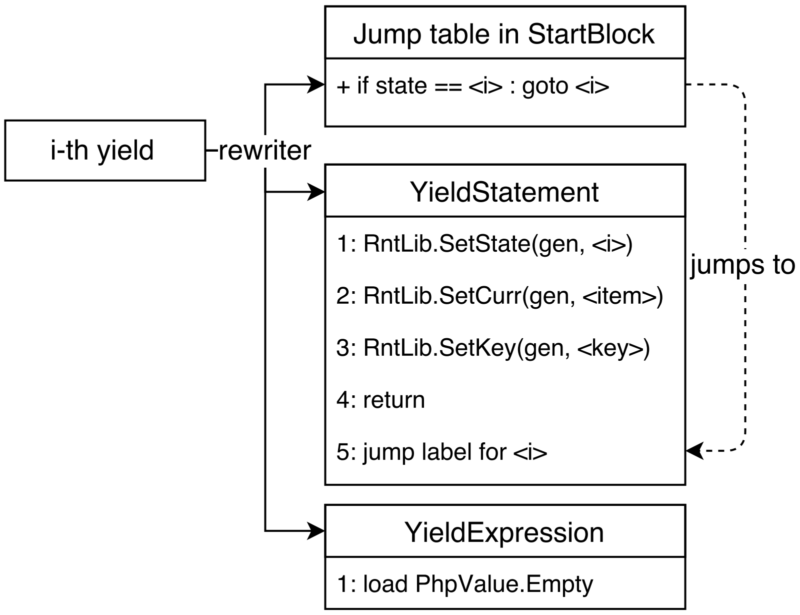


Figure 5.3: Splitting of a yield into an expression and a statement.

Second, we can cut any branch in an expression tree and prepend it before the tree while keeping the meaning of the program the 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 (Figure 5.4). 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 of the captured branch and that were therefore supposed to be executed first. In the figure below, the expressions 5 and 6 respectively will get executed first, even though they should come after expressions 1, 2, 3, and 4.

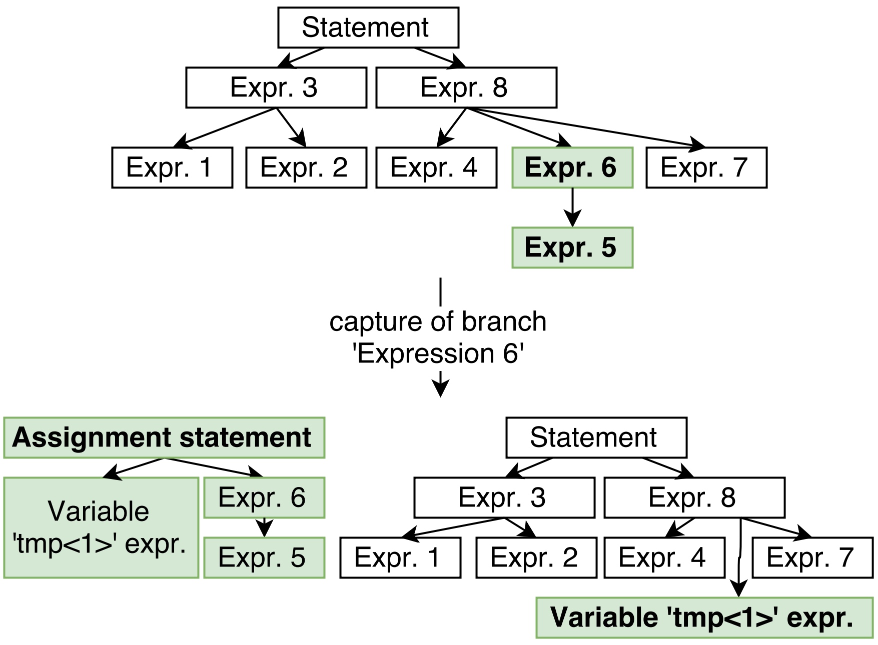


Figure 5.4: Capturing a sole branch.

An obvious solution to this problem is to cut and prepend not only the one branch we want to capture, but also, in their respective order from left to right, all other branches that are supposed to get executed before it (Figure 5.5). Since the semantic graph emit, and thus also the execution, follows a post-order traversal, we must cut and prepend all branches that are higher and to the left of the branch we want to capture.

To be specific, that includes all branches that start to the left of the path between the root of our captured branch and the root of the whole semantic graph. The reason lies in a post-order traversal of the semantic graph.

It starts with the graph's root. Then it goes through the root's leftmost child, followed by its next child, and so on. Let us say, for example, that the second element of our aforementioned path is the root's third child. When the traversal enters it after going through the branches started by the root's first two children, it goes into its leftmost child first, again. It then continues the same way until it encounters the root of our branch. When that happens, it keeps following the same logic, traversing the whole branch before closing its root and starting to visit any other nodes. After the traversal is finished with the branch, it closes its root and goes one level up, starting to traverse the branch's root's first sibling to the right.

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 cut and prepended. The ones on the path have our branch among their children and thus need its result - our branch - to be evaluated first. And the ones on the right need to be evaluated later, simply because of post-order traversal rules.

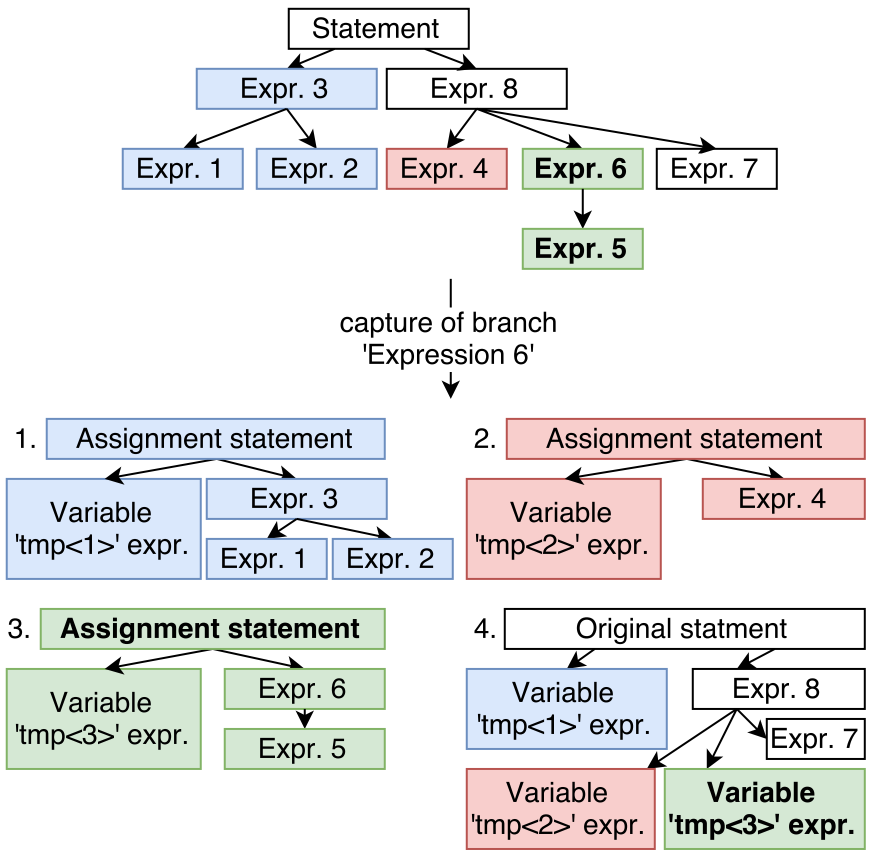


Figure 5.5: Capturing a whole branch while maintaining execution order.

5.2.3 Semantic tree transformation

Now we can combine the two observations from the beginning of the previous section and come up with a solution to our *yield* as an expression problem. Our goal is to split each *yield* and then separate its statement part while maintaining the original order of execution.

To be able to do it, the *yield* has to be on top of an expression tree first. It must be, in terms of execution order, its first user code representing node. When that is true, we can split the *yield* into a *yield expression* and a *yield statement*, and then put the statement part before the tree.

This does not change the program's meaning or order of execution, because the *yield statement* is directly followed by the tree and the tree's first user code node is the *yield* expression. And since the part of the *yield* containing a *return* is a proper statement before - not inside - a tree, the semantic graph is actually emitable.

With that, our problem has shifted to transforming expression trees containing *yield*s at arbitrary places into multiple expression trees that have them only as their first user code representing nodes. If we were able to do that, we could split each of the *yield*s and separate their *return* containing parts as standalone statements.

The transformation can be done through our branch capturing mechanism. We can take each branch that starts with a *yield* and capture it (Figure 5.6). That prepends its tree with an assignment of the captured branch and all other branches that are, within the tree, supposed to get executed before it. None of the other branches are of significance for us, so let us ignore them and focus only on the captured one.

The tree representing a prepended assignment of the capture branch contains a *yield* as its first expression that is not the synthesized *assignment statement* itself. This holds true, because we specifically captured a branch that starts with a *yield*. As such, the tree representing the prepended branch fully adheres to our requirements for splitting the *yield* into a statement and an expression.

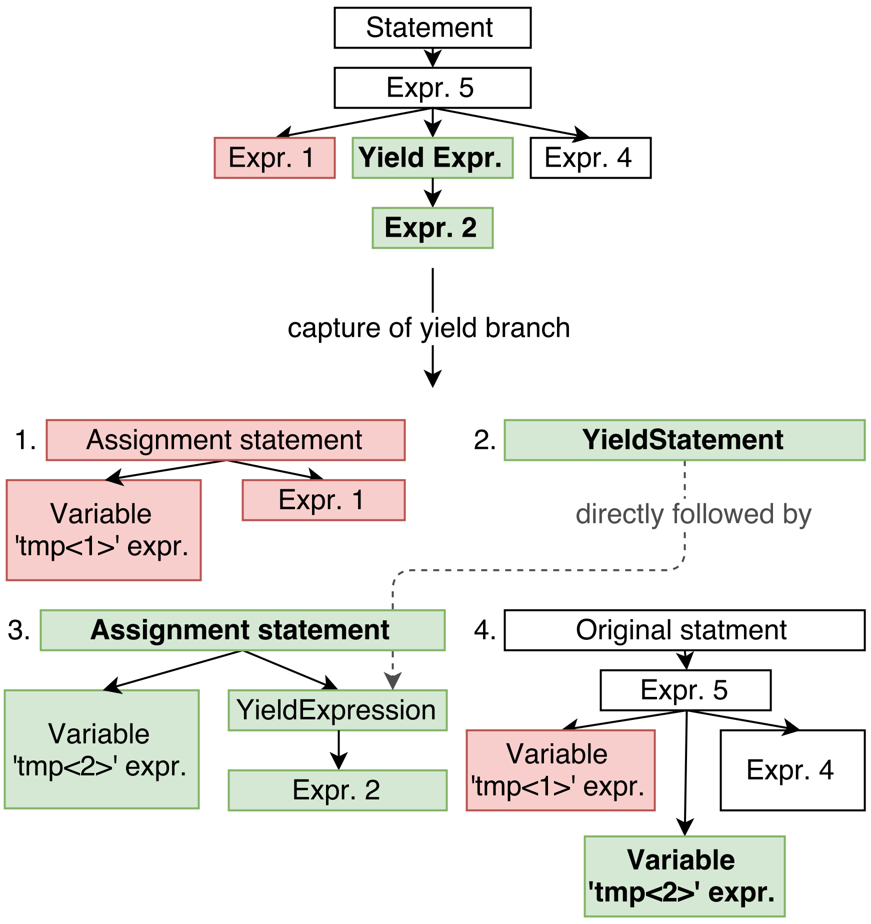


Figure 5.6: Capturing a whole branch starting with a yield.

Now we have to repeat the aforementioned process for each expression tree containing *yield*s and for each of their branches that starts with them. Every run of our algorithm fixes one *yield* and results in a correct semantic graph that can serve as a starting point for another run. This means that after a finite number of repetitions, all *yield*s will have been split while keeping the original program's meaning and execution order intact.

5.2.4 Short circuit evaluation

The solution from the previous section would work perfectly, if it were certain that all branches of an expression tree will always get evaluated. The problem is that they do not. For example, a conditional operator *? :* is represented by an expression that has three children, only two of which get evaluated in any given situation. Its children are a *condition expression* that gets evaluated first and every time, and two *conditioned expressions* representing its return value. One that gets evaluated and returned when the condition turns out to be true and one for when it is false. A similar situation transpires with short-circuit evaluated binary expressions such as coalesce, and, and or operators.

The problem is that a *yield* can be inside one of these *conditioned branches* and when its gets captured and prepended before the tree, it will always get executed regardless of the *condition*. For example, even if the *condition* were false, an expression tree for its true branch, being prepended before the tree because it includes a *yield*, would still get executed and had its result saved into a temporal variable. The variable would simply not get used as a result of the *condition expression* (Listing 7).

```php
<?php
// Original expression before capturing the yield branch.
$result = isTrue() ? yield 0 : "falseBr";

// Expression with a captured yield branch.
$tmpBranch = yield 0; // The yield is evaluated everytime.
$result = isTrue() ? $tmpBranch : "falseBr";
```

Listing 7: Conditional expression whose captured branch is not conditioned.

This, of course, represents a problem. While the result of each expression would remain correct, a whole expression branch would now get executed every time instead of only when a certain condition was satisfied. And even if the branch itself did not have any side effects, it starts with a *yield* which includes a *return*. A *return* that was previously conditioned but is now executed every time.

Fortunately, the solution is relatively simple. When going through the semantic graph to capture branches starting with a *yield*, the mechanism can remember when it enters and leaves a *conditioned branch*. And when it is about to capture a sub-branch while being in a *conditioned branch*, it can simply not prepend its assignment as described previously. Instead, it can prepend a *condition edge* that has the *assignment statement* in its true block and a logical conjunction of all conditions guarding the current branch as its *condition expression* (Listing 8). That way, the prepended assignment with the captured branch will get executed only when the condition is true, in essence in the same situations in which it, at its original position, in the tree would.

```php
<?php
// The condition `isTrue()` is evaluated twice.
if isTrue() { $tmpBranch = yield 0; }
$result = isTrue() ? $tmpBranch : "falseBr";
```

Listing 8: Conditional expression whose condition is evaluated twice.

Unfortunately, that in itself is still not enough. With that reuse, each *condition expression* is possibly evaluated multiple times. Once as the original expression in the tree and once for each *condition edge* created by capturing a sub-branch from the *conditioned branch*. Even this problem has a straightforward solution.

One can capture each branch representing a *condition expression* whose respective *conditioned branches* are to be captured in the future, i.e. that include a *yield*. Subsequently, one can use the temporal variable created by the capture instead of the expression itself in both the *condition edge*s and its original location in the tree (Listing 9). That way, the *condition expression* gets evaluated only once as part of a prepended assignment and its result is reused. And because the capturing also prepends all branches that are before the *condition expression* in the tree, the order of execution stays the same.

```php
<?php
$tmpCond = isTrue(); // Condition expression is evaluated once.
if ($tmpCond) { $tmpBranch = yield 0; }     //`tmpCond` is reused.
$result = $tmpCond ? $tmpBranch : "falseBr";//`tmpCond` is reused.
```

Listing 9: Conditional expression captured correctly.

5.3 Yield as an expression - implementation

In previous chapter we have described an algorithm to solve our *yield* as an expression problem, including edge cases such as conditioned branches. Now we will talk about how the algorithm can be, and in fact is, implemented within the Peachpie compiler. Before going into details, however, we will first take a look at how the sent value actually gets into the generator to be later used as the *yield expression*'s value.

As mentioned in the PHP generators chapter[5], the value gets in through a *send* method defined by an *Iterator* interface. This method takes the value as its first argument. A simple way to implement it on the generator is to add a backing field on it and in the method assign the sent value to the field. Then, the expression part of a *yield* can emit itself as a read from said generator's field, leaving the sent value on the evaluation stack.

With that out of the way, let us move to the algorithm implementation. We need to implement a transformation of a semantic graph. As with the previously described implementation of limited generators, there are two ways to go about it. The transformation can either be done by a standalone rewriter component or

[5]Chapter 1.3

it can be integrated within the *SemanticBinder* and done as part of the binding phase.

While the standalone approach might be better in terms of architecture, is has several downsides. First, a support for rewriters would have to be introduced within Peachpie. For that we would need, among other already mentioned things, to be able to traverse and modify the semantic graph in a generic way, which is something, we cannot currently do. There exists a base visitor, that can go through the graph but it requires specific knowledge about each node to visit its children. So, if we did not want to implement the transformation for each node separately, we would need to add a generic way to query and replace a node's children.

Second, having a rewriter would necessarily introduce an additional traversal of the tree. Despite the fact that the performance penalty would not be big, after all it would happen only for generator methods, it was still a factor that contributed towards pursuing the other option instead. To do the transformation during a binding phase.

It might not seem at first, but the *SemanticBinder* is actually an ideal place for implementing the transformation algorithm. It builds the graph in a post-order, has full knowledge about every semantic node, and contains a method that creates each node of every expression tree. While it might not be clear now, all of these properties will become useful.

The idea behind our implementation is following: when the *SemanticBinder* is asked to create a semantic representation of an AST, it does not return just a bound expression tree. Instead, if the AST contains a *yield*, it returns an already transformed forrest of expression trees with separated *yield statement*s in between.

In fact, saying it returns an already transformed forest is not technically correct. There is no transformation. All the branch capturing and conditioned expressions handling is done directly when the respective semantic nodes are created by the semantic binder. There is no single expression tree created first and then transformed, the correct forrest gets created right away.

5.3.1 Binding multiple elements

To support the transformation algorithm within the *SemanticBinder*, we need to enable it to return multiple trees that represent the captured branches first. Specifically, in addition to either a bound statement or an expression it always needs to be able to return an arbitrary semantic subgraph that is supposed to go before the currently bound element. We will call it a *pre-bound graph.*

A whole semantic subgraph, which means bound blocks connected with arbitrary edges (Figure 3.3), is required instead of just a list of statements, because some of the prepended statements might need to be conditioned by a *condition edge.* This is the case for, for example, assignments created by capturing branches from short-circuit evaluated binary expressions.

While it would be possible to create something like a *conditioned statement* and use that, there is an obvious advantage to using an already existing mechanism. All other modules, such as a flow, diagnostics, and type analysis, know how to handle a *condition edge.* If we went with a new custom statement, however, we would need to support it in all of these ourselves. Also, creating a new node for something that can be expressed with existing ones is not very clean architecturally.

To enable returning a full subgraph, we need a container for the bound element and the graph itself, first. It can be a simple structure holding either a bound statement or a bound expression and two references to bound blocks (Figure 5.7). Two because, as we will see, we need both the first and last block to properly connect the *pre-bound graph.* Let us call them *first* and *last pre-bound blocks* and the whole container a *bound bag.*

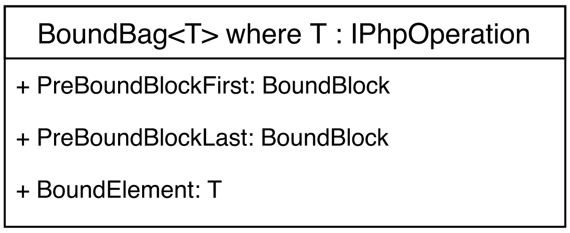

BoundBag<T> where T : IPhpOperation
+ PreBoundBlockFirst: BoundBlock
+ PreBoundBlockLast: BoundBlock
+ BoundElement: T

Figure 5.7: Bound bag.

With the container, we need to change the public public API of the *SemanticBinder* so that it exposes methods that return this *bound bag* instead of a plain bound expression or bound statement. We cannot replace the original methods, however. They are not used just externally but also internally by the *SemanticBinder.* It uses them to bind individual nodes one by one when it is asked to bind a whole expression tree. And in these situations we want them to return plain old bound elements. More about that later.

Instead, we can make the original methods private and add new public methods, let us call them *BindWholeStatement* and *BindWholeExpression*, that will return a *bound bag.* For now, they can call the private ones and only wrap their results into *bound bags* with empty *pre-bound graphs* (Figure 5.8). We will get back to them later too. Before that, however, we need to fix all the external calls to the - now private - methods and change them to accept a bound bag instead of a sole bound item

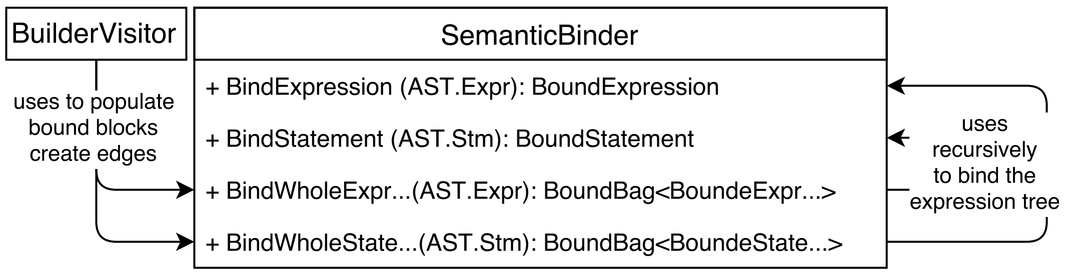

BuilderVisitor
uses to populate bound blocks create edges
SemanticBinder
+ BindExpression (AST.Expr): BoundExpression
+ BindStatement (AST.Stm): BoundStatement
+ BindWholeExpr...(AST.Expr): BoundBag<BoundeExpr...>
+ BindWholeState...(AST.Stm): BoundBag<BoundeState...>
uses recursively to bind the expression tree

Figure 5.8: Builder visitor's and semantic binder's relationship.

The fix is straightforward for statements. The original *BindStatement* method is externally called just once, when the *BuilderVisitor* needs to bind a new statement to add it to its *current bound block.* To support getting a *bound bag* instead of plain bound statement we need to connect its *pre-bound graph* before adding the bound statement to the *current block.* That means creating a simple edge between the *current block* and the *first pre-bound block*, and setting the *last pre-bound block* as the new *current block* (Figure 5.9). As the one, into which the bound element will get added.

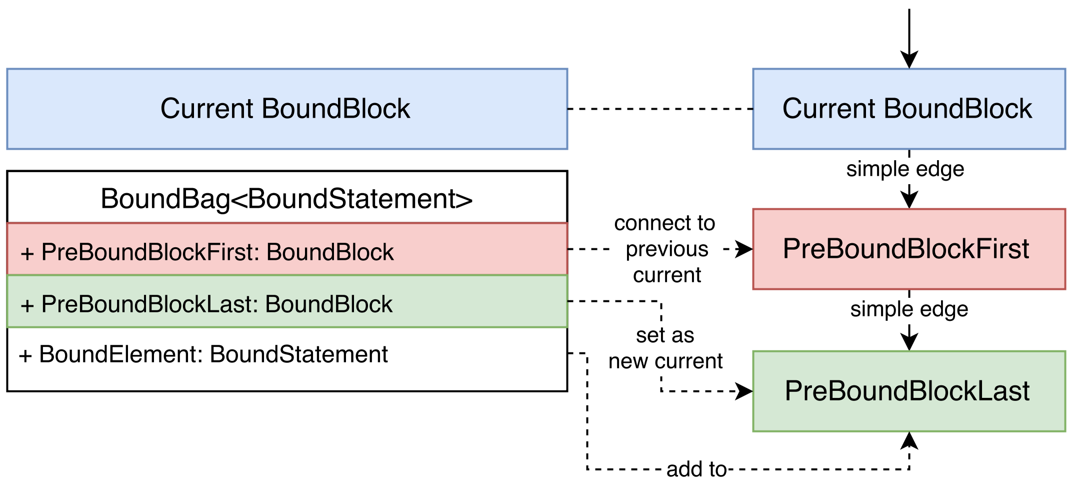


Figure 5.9: Connecting a bound bag as a new statement.

The situation regarding expressions is a bit more complicated. *BindExpression* is externally called in three situations. When another symbol needs to bind a constant, for example a parameter its initializer. When a *BuilderVisitor* needs an expression representing a reference to a synthesized but user accessible variable, such as *foreach*'s key and value variables. And lastly, when also a *BuilderVisitor* needs to bind an expression for some edge, like a *condition expression* for *condition edge*.

The first two cases are relatively easy to adapt to the bound bag. Both variable references and constants are guaranteed to not to include a *yield* and thus to not to produce a *pre-bound graph*. Therefore, we can retrieve the bound expression from their bound bags and leave the implementation without any further modifications.

The third one is a bit more complicated. There are four instances in which the *BuilderVisitor* asks the *SemanticBinder* to bind a full unrestricted expression. For a *switch edge*'s switch and case values, *condition edge*'s condition, and *foreach*'s enumeree. All of them are problematic because when new edges are being created, it is hard to say what the *current bond block*, to which the *pre-bound graph* should be connected, actually is.

Generally, there are two possible answers for that. One is enriching each of these edges with special branches for these expressions' pre-bound blocks. On one hand, this is more systematic, it clearly maps the intend to the semantic structure. On the other, it increases complexity for all of these edges' instances to support a feature used only by generator methods. The second way is to choose any appropriate block and connect the *pre-bound graph* to it. An appropriate block means any block that the *pre-bound graph* can be connected to with the condition that emit of the result must produce a program with correct PHP semantics.

For *condition edge*'s condition, *foreach*'s enumeree, and *switch*'s switch value expressions one such block is their edge's *source block*. That it the block, the edges connect to as their starting point and also usually the last *current block*.

It is a good choice, because these expressions are the first things executed within their respective edges and so the block directly before said edges is also a block directly before them.

Thus, for these three the solution is to take the *first pre-bound block*, connect it to the edge's *source block*, set the *last pre-bound block* as the new *source block* (Figure 5.10), and continue the edge creation using a bound expression as before.

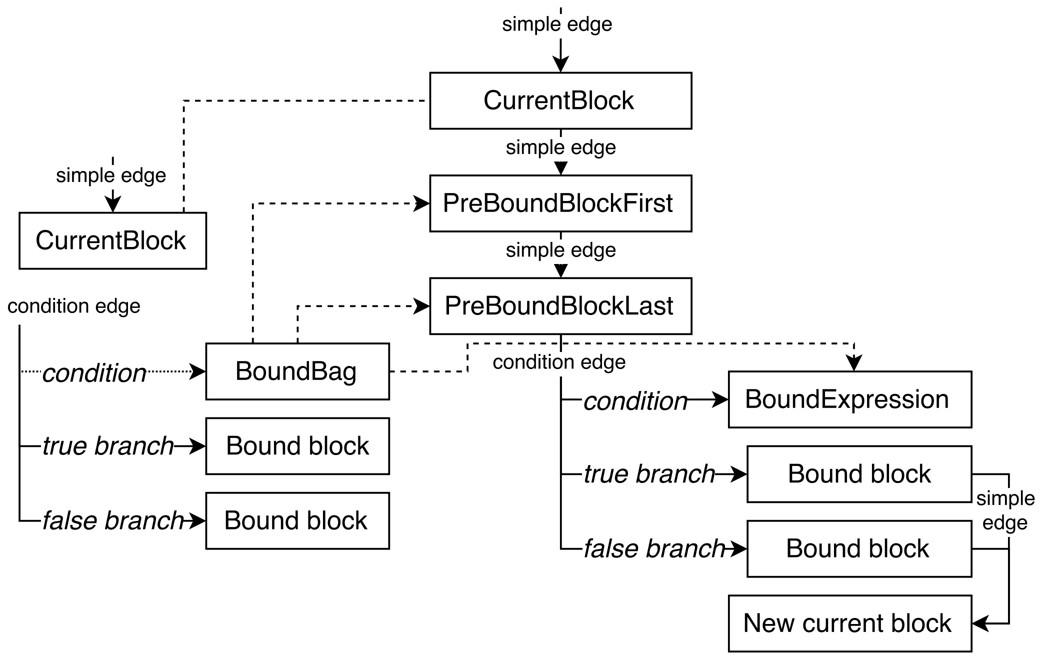


Figure 5.10: Connecting a bound bag as a condition edge's condition.

This approach is, unfortunately, not applicable for the fourth case, a *switch edge*'s (Figure 5.11) case value expression. This is an expression representing the value a *switch value* is compared to for each case block. The reason is, that the *case value expression* is executed neither in the beginning of the edge or directly after some other block, so there is simply no block to connect the *case value*'s *pre-bound graph* to. With it we need to go the first way and implement a support for *pre-bound blocks* directly on the edge. Actually, not on the edge itself but on the *case block* the *switch edge* points to.

That itself is not that complicated. We can remember the whole case value bound bag in the case block and emit the *pre-bound graph* directly before the case value condition. The issue is, that the evaluation stack is not guaranteed and actually never is empty at that point. Because only one case block can be taken in PHP switch, Peachpie actually keeps the *switch value* on the evaluation stack from the beginning of a switch edge evaluation and goes one *case block* after another comparing their *case values* with it.

Therefore, if a particular case block has any *pre-bound graph* we need to pop the *switch value* first and then, after emitting its *pre-bound graph*, load it again. To not to evaluate the *switch value* multiple times, we also must replace it with a temporal variable. A temporal variable into which the *switch value* is saved, or captured in other words, in the very beginning.

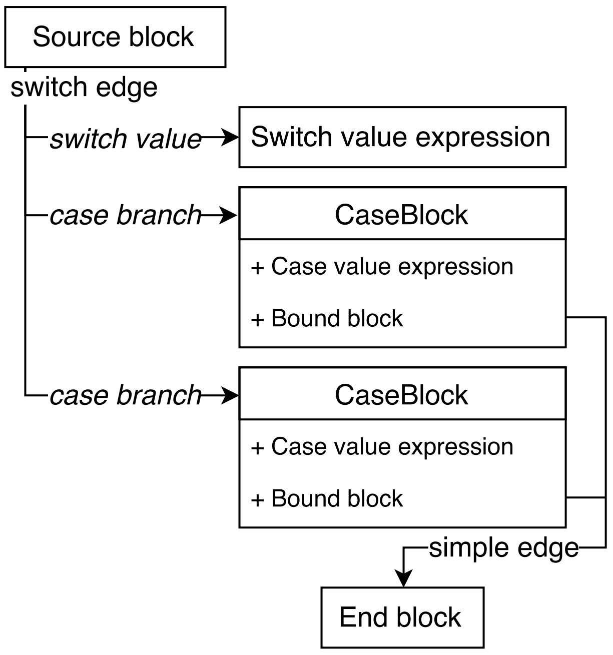


Figure 5.11: Switch edge diagram.

There are certain optimizations that can be done as part of this process. For example, we do not need to do the whole switch value saving if it is a constant. They are, however, implementation details.

5.3.2 Capturing branches with yields

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 binder is asked to bind an AST, be it an expression or a statement, it first of all prepares an empty *first pre-bound block* for the item that will be bound. Then, it proceeds with binding the element, its children, and so on recursively as already described in the semantic graph chapter. And when any of these nodes is under any path between a *yield* and the root of the currently bound expression tree, it creates its semantic representation but does not simply return it. Instead, the semantic binder constructs a new *assignment statement* that sets the bound expression to a new temporal variable, puts said statement into the current *pre-bound block*, and returns a *read expression* from the temporal variable (Figure 5.12). That way the read from the temporal variable ends up in the resulting expression tree instead of the bound expression and the bound expression becomes captured. And when the AST finishes binding it gets returned as part of the bound bag's pre-bound graph.

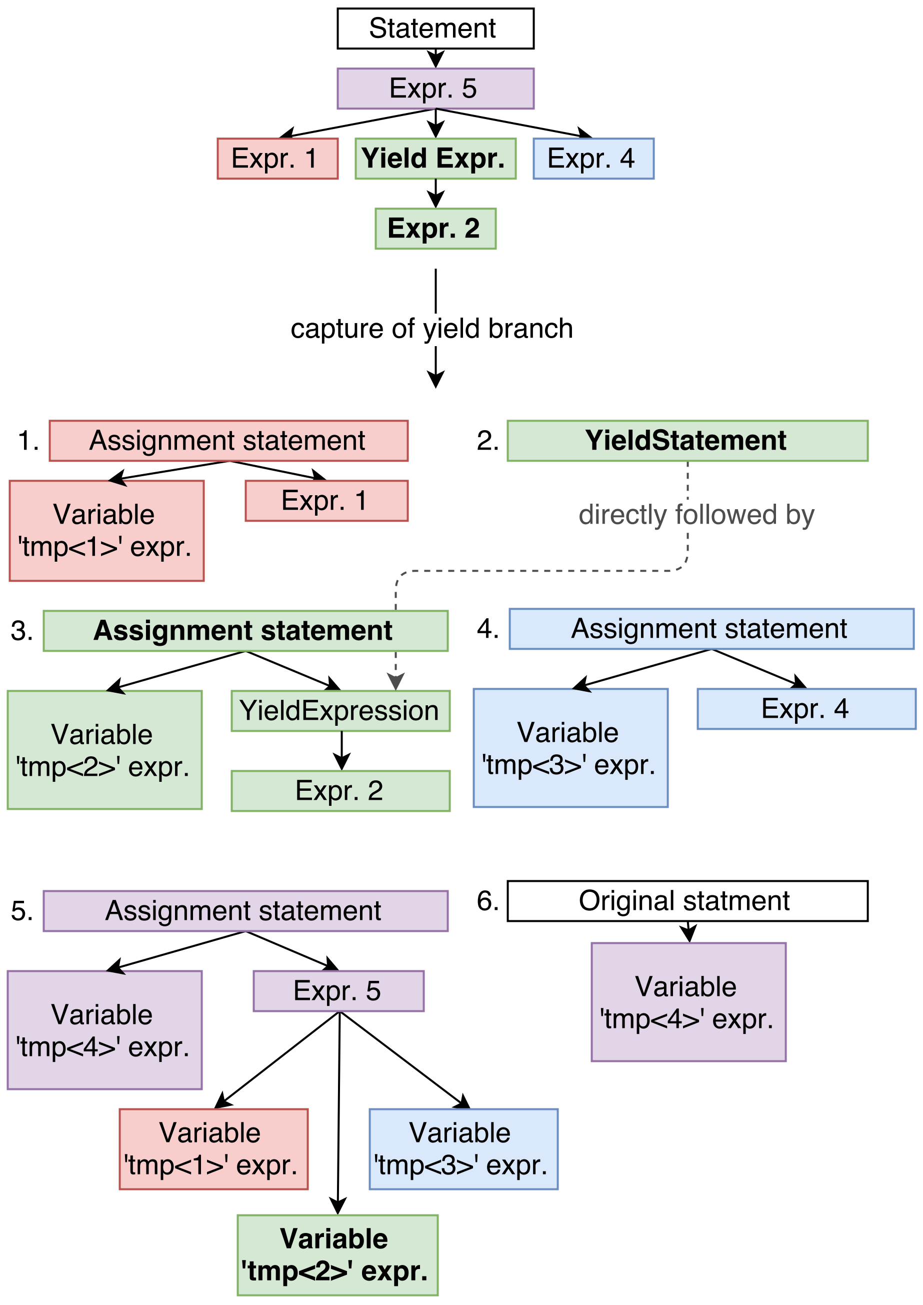


Figure 5.12: Capturing a branch with a yield.

Two things are needed to support such behavior, the ability to determine whether any particular node is under a path between the current expression tree's root a *yield* and a way to keep *pre-bound blocks* while an expression tree for an AST is being binded. Before going into details on implementing each of these, however, let us first take a look at a small modification we have made to the algorithm in comparison to the one described in a previous chapter[6].

5.3.3 Correctness of modified capturing algorithm

The actually implemented algorithm captures all branches that start with children of elements on the *root-yield* path instead of only the children to the left. It also

[6]Chapter 5.2

captures branches starting on the path itself, after all they are children of some elements one level higher in the path. The reason for that is, that neither the AST nor the semantic tree has a generic notion of left to right ordered children. Therefore it would be hard to determine whether a current element is to the left of the path or to the right. This change, however, does not alter the fact that the transformation keeps the original order of execution.

The implemented version captures a superset of branches the theoretical algorithm did. The ones on top of that are, as the original were, still captured and prepended in a post-order, which guarantees the correct order of execution. In essence, the ones to the right will get prepended and thus executed only after those to the left, and the ones on the path itself only after all their children.

The capturing is done in a post-order order because following statement holds true. The *SemanticBinder* itself operates as a depth first search and the capturing is the last part of a node's binding process. When an element is being bound, its semantic representation is created first, part of which is fully binding its children from left to right, and only after that, the node itself might get captured and subsequently returned as bound. The thing is, that as part of the children's' binding process, they themselves might get captured and therefore prepended first.

Capturing both a child node and its parent's, which is something that happens with our version for almost all elements on the path, is not a problem either. It creates two prepended statements, first the child's and then the the parent's as proven above. The parent's prepended statement will contain an expression tree with the children's original branch replaced by the children's temporal variable (Figure 5.12). And the original expression tree will contain the parent's branch replaced with the parent's temporal variable. Since the children's prepended statement is first, it will get executed before the parent's. Thus the execution order will remain correct.

5.3.4 Creating and keeping the pre-bound graph

Now that we are sure our modified version is still correct, let us go back and implement the two things we need for the algorithm to work. First, we will take a look at creating and keeping the pre-bound blocks while binding an AST.

The keeping part is straightforward, we only need two new fields with references to bound blocks on the *SemanticBinder*. Two because, as we have discussed before, we need both the first and the last block for the eventually returned *bound bag*. The reference to the last bound block can also serve as the *current pre-prebound block* into which new captured branches are added.

The actually complicated part is getting new empty bound blocks. While the *SemanticBinder* could just create brand new bound blocks instances itself, there is a small problem with the fact that bound blocks are in fact ordered. They all have an ordinal number that represents where - in comparison to other bound blocks - do they fit within the original source file. The higher their number the further the code they represent was in the original source code. For example imagine an if statement (Figure 5.13): the block before the if itself must have the smallest ordinal number, the true block higher, the else block higher still, and the block directly after the highest.

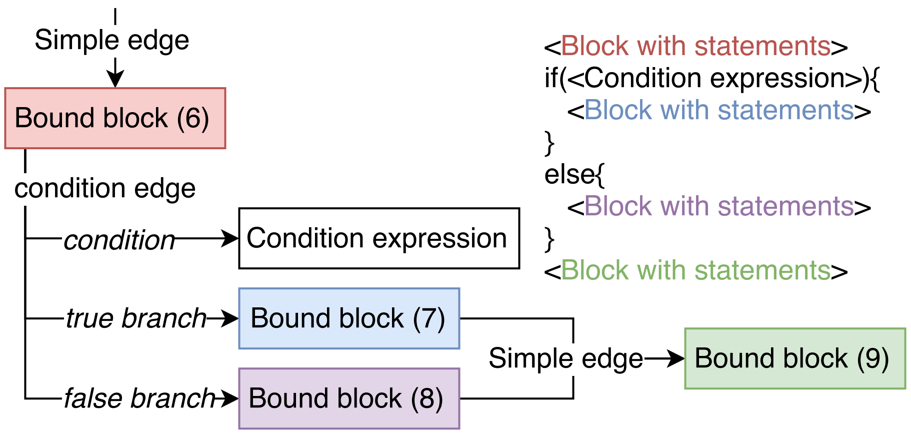


Figure 5.13: Ordinal number of bound blocks.

The problem is that the *SemanticBinder* does not know the last ordinal number the *BuilderVisitor* used to create its *current bound block*. And so, it does not know what ordinal number to use for a new *pre-bound block* that will represent the current statement's or expression's *pre-bound graph*.

It is true that the *SemanticBinder* could create all these *pre-bound blocks* with an ordinal number 0 and then the *BuilderVisitor* could fix them as part of the phase that connects the *pre-bound graph*. That would, however, require creating a new component that could traverse the graph and would know its topology. While the *BuilderVisitor* knows how to create the semantic graph, it does not have generic tools to traverse it or to update its nodes' ordinal numbers.

It was also briefly considered to simply pass the last used ordinal number with each to call the *SemanticBinder* and then return the new one as part of the *bound bag*. This approach was not chosen in the end, however, because it breaks the abstraction separation between two relatively independent components.

Thus, it was chosen to offload the creation of new bound blocks to the *BuilderVisitor* instead. As part of an initialization phase, the *SemanticBinder* takes a reference to the *BuilderVIsitor* and, when needed, uses it to get new bound blocks with the correct ordinal number. The only requirement for this to work is to ensure following invariant stays true. When an element is being bound, the last block created by the builder visitor must be the block directly before the element. And as it turns out, this is an already existing invariant in the Peachpie compiler.

With this approach, the *SemanticBinder* can get a new empty bound block with the correct ordinal number without having to actually know or care about ordinal numbers. For now, this is relevant only for the one initial bound block that gets created in the beginning of a binding process, is used as the storage for all the captured branches, and in the end gets returned as part of the *bound bag*. Later, it will become more important when conditioned branches come into play because they will require getting a number of bound blocks for the *pre-bound graph*.

5.3.5 Path between the root and yields

To be able to tell whether an element is under a path between the root of the current expression tree and any *yield*, we need to have the paths first. While it would be possible to simply traverse the full graph with a depth first search and record the path whenever we find a *yield*, there is actually a simpler way.

As part of the AST the parser returns a list of references to all found *yield expressions*. Due to that, we only need to go through this list, assign individual *yield*s to their respective method symbols, and realize these paths. The first part of the process are trivial busywork and thus will not be discussed in detail.

Since the paths are needed before any node from a method is bound, the realization is best done as part of the method's *SemanticBinder*'s initialization. The realization itself is a two steps process. First, the abstract syntax nodes representing the *yield*s must get into the *SemanticBinder*. They can simply be passed there from the routine that creates the method's CFG and thus also initiates its *SemanticBinder*.

The second step is actually creating the paths. For that, it is useful to know that all syntax nodes have a generic reference to their parent node. Thus, it is quite easy to traverse the syntax graph upwards, which is all we need to do. We can start with every *yield expression* and just go upwards in the graph until we get to a node representing the enclosing method, recording the path along the way. Do note that paths created this way are not the paths between a *yield* and its expression tree's root but a *yield* and the method's root. While it might look as a minor thing it will be important.

Now that we have the paths for all *yield*s within a method, let us take look at the mechanism to find out whether a node is under any of said paths. Just checking whether the parent of a currently bound abstract syntax node is within any recorded path is not enough. It is not enough precisely because the paths do not stop at expression tree roots but instead continue to the methods' roots and as such include things like edges.

For example (Figure 5.14), if a *yield* was in a true branch of an *if edge*, the path would also contain the *if edge*'s syntax node. Therefore, the first node of the *if edge*'s *condition expression* would report as being under the path despite the fact it actually is not under the path we care about, the path between a yield and its expression tree's root.

Therefore, we need a more nuanced mechanism. A good observation to make is, that we know about each expression tree's root. It is the node passed to the public *SemanticBinder*'s methods that invoke the binding process of an AST, the *BindWholeExpression* and *BindWholeStatement*[7]. With this knowledge, and the fact that the binding process is essentially a depth first search, we can solve this issue through a new *level* field on the *SemanticBinder*. A *level* field that represents how deep we are under the closest *yield-root* path in the currently bound expression tree (Figure 5.14).

[7]Chapter 5.3.1

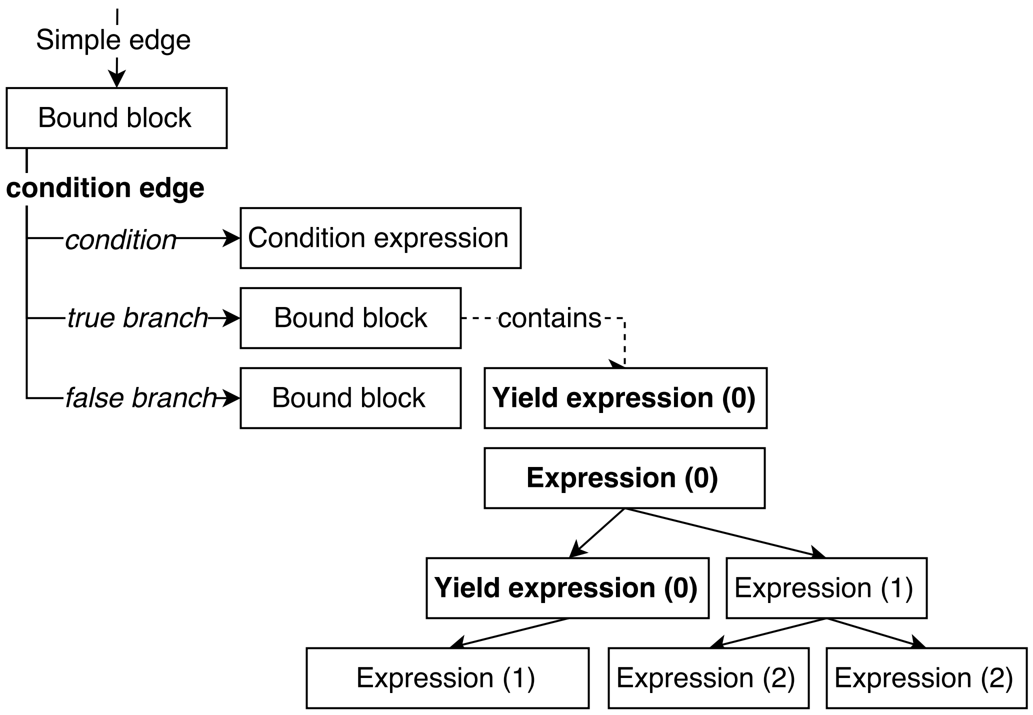


Figure 5.14: Path between a yield and expression tree's root.

When the *SemanticBinder* starts binding an AST through any of its public methods, the *level* is set to -1, representing that there is no known relevant path above. There cannot be one because at that point the current node, that is being bound, is the expression tree's root.

Then, in the beginning of each of the methods that bind the tree's individual nodes, *BindStatement* and *BindExpression*, the *level* is saved to a local variable and the field is updated. The update consist of two operations. First, if the current *level* is not -1, it is increased by one, indicating that we are one level deeper. Second, it is checked whether the current node's parent is part of any path and when it is, the *level* is set to 0.

After setting the *level*, a semantic representation of the node is created, and its children are bound, potentially also captured. Then, the current *level* is checked. If it is 1 or 0, which means the node is either on the path or directly below it, the just bound node gets captured as described before[8], adding a statement that assigns it to a new temporal variable into the *current pre-bound block*. Finally, the original *level* is restored from a local variable and either the bound node or, if it was captured a read from its respective temporal variable, gets returned.

5.3.6 Conditioned branches

With that, we have all the building blocks needed to have an implementation that handles yields as expressions in every situation except for when the yield is in a conditioned branch. That means either a branch of a *conditioned expression* or a short-circuit evaluated *binary expression*.

[8]Chapter 5.2.2

For a *conditional expression* there are two *conditioned branches* and one separate *condition branch* that dictates which of the first two branches will get evaluated. For short circuit evaluated *binary expressions* there only two children branches in total. A *condition* one and a *conditioned* one. The rule is that it the *conditioned* is evaluated only if the first one, the *condition* one, has not already satisfied the rule of the short circuit evaluation.

We have already discussed the theoretical approach to solving this problem[9]. Its implementation is following. When binding an element with a conditioned branch, the branch is bound as a standalone expression tree. The result of the branch's binding, its *bound bag*, then used to determine whether the branch contains a *yield* or not. The key is whether its *pre-bound graph* is non-empty. When it does contain a *yield*, said *pre-bound graph* is connected to the *last pre-bound block* of the currently bound expression tree through a *condition branch.* A *condition branch* that uses the same *condition expression* that guards the branch (Figure 5.15). This effectively ensures the the *pre-bound blocks*, constituting the *pre-bound graph*, of the branch are evaluated only when the branch is, in essence when its condition is true.

The issue with multiple evaluations of the *condition expression*, that can happen because now we have the condition both in the original expression and in the *condition edge*, is solvable through a simple observation. When the *conditioned branch* includes a *yield*, the expression that the branch stems from must be on the path between it and the expression tree's root. Therefore, the *condition expression*, that starts from the same node the *conditioned branch* does, is guaranteed to be directly under said path.

With that knowledge we can capture the *condition expression* while binding the shared parent node and reuse its temporal variable in the both the expression's condition, i.e. the original position, and the branch's *pre-bound graph*'s *condition edge.* Since the *condition expression* is guaranteed to be under the path, all expressions to the left from it has already been captured and prepended and so the order of execution stays correct.

[9]Chapter 5.2.4

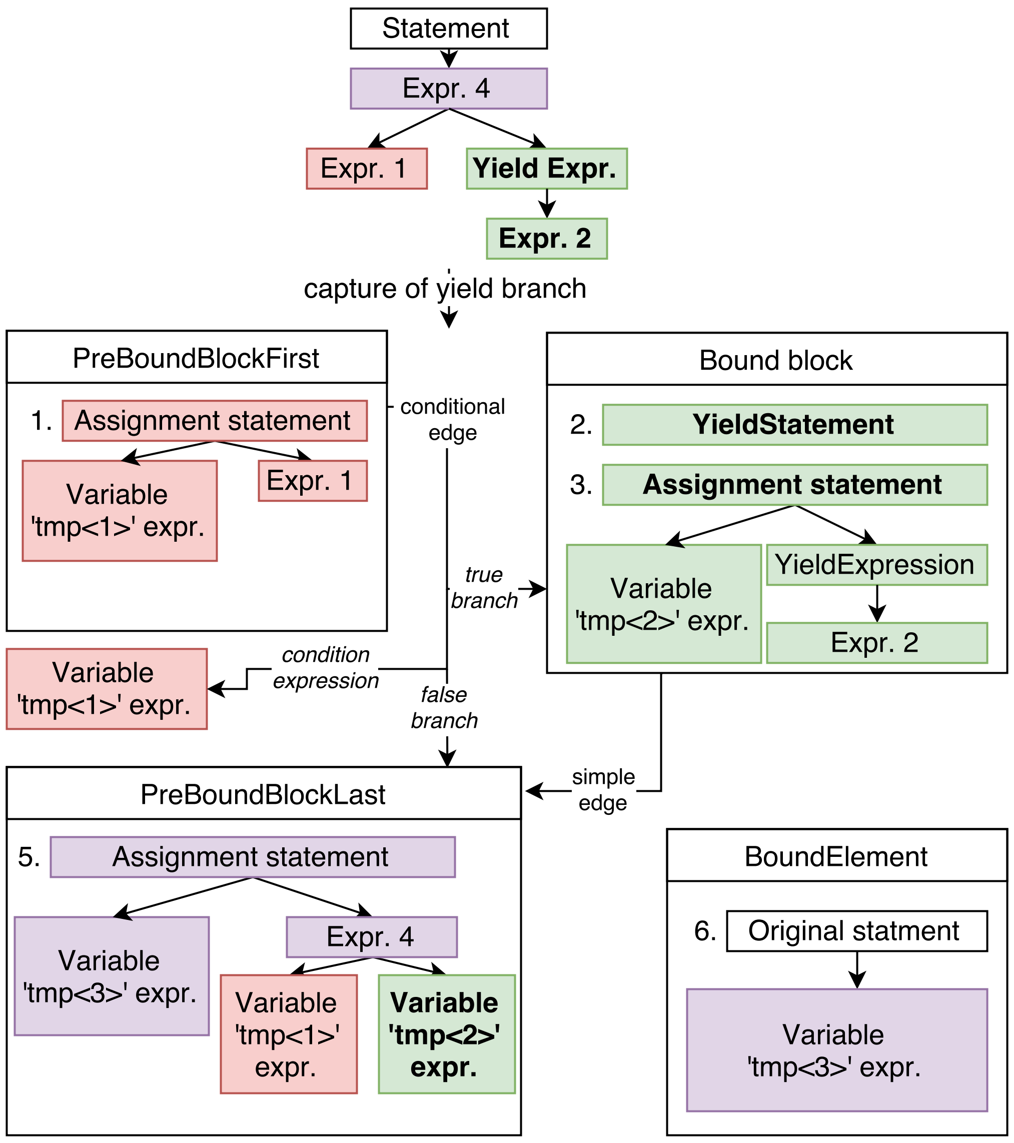


Figure 5.15: Capturing a yield in a conditioned branch.

Now the only thing that remains to be described is the way to bind the branch as a standalone expression tree in the middle of binding its enclosing one. One possible way is to save both references to the *pre-bound blocks* on the *SemanticBinder* to local variables, initialize them to a default value, and proceed with binding the branch through normal means. When it is done, capture the branch's *pre-bound graph* from the now re-populated *pre-bound block*s references, construct its bound bag, restore the *pre-bound block*s references to their original values, and return the bag. That way the solution works even with nested conditioned branches.

5.3.7 Implementation remarks

In previous sections we have presented a high level discussion about the implementation of an algorithm that lowers generator methods using only relatively small modifications to a *SemanticBinder* and a *BuilderVisitor*. While they are more of implementation details, there are a few remarks that could be useful for the reader but do not fit into any previous chapter due to either being too specific or requiring knowledge about the system as a whole.

Not all expressions need to be captured when the algorithm signals they should. Since the capture of non-yield branches only ensures the correct order of execution, it is redundant for expressions that have no side effects and whose value cannot change in time. While these properties are hard to prove for an arbitrary expression they are guaranteed to be true for two specific types of expressions: a *constant expression*, such as a number literal, and an expression that has a constant value resolved by the compiler. Thus, branches consisting of only these types of expressions can be omitted from capturing.

To mitigate any possible performance penalties for non-generator methods and separate generators specific code, it is possible to have two semantic binders. One that serves as the basis and is used for all normal methods and the other that inherits from it and handles generator methods.

The only change we need to make to the basic one, in comparison to the original pre-generators *SemanticBinder*, is to create the new public methods for binding expressions and statements and make them return *bound bag*s with empty *pre-bound graphs.* In the generators specific *SemanticBinder* we can override both these methods and the ones that bind the AST's individual nodes and put all the generators specific logic there.

The question which *SemanticBinder* to use for any particular method has a straightforward answer. Since a *SemanticBinder* for a method is initiated only after the method's AST has been created by the parser, it is already known whether the method contains any yields and as such if it is a generator method. With that knowledge we can simply create the appropriate *SemanticBinder* and pass it to the method's respective *BuilderVisitor* to use.

There is obviously more to the implementation than described in the chapters above. To name a few, we have completely left out the changes required to make return type analysis work, complexities related to throwing into and rewinding generators, all logic associated with generator's return value, or how Peachpie keeps the list of method's yields to be able to construct the jump table in the beginning of a generator's *next* method. All of these are, however, details that might, and most probably will, change in near future as the Peachpie project matures. Thus they do not provide any value for a reader that might want to either understand why peachpie works a certain way or implement a similar feature. And for those that are interested in line by line explanation the source code, or at least the part implemented as part of this thesis, is commented in detail.

5.4 Yield in exception handling blocks

Now that we have fully described the implementation of generators within the boundaries set in the introduction, let us briefly look at possible ways to expand them. Specifically, at dealing with with *yield*s in exception handling blocks. Due to the extent of this thesis and the complexity of arbitrarily placed *yield*s[10] we will neither propose a full solution nor describe an actual implementation. Rather, the

[10][Lippert, 2009a, **?**,c]

following chapter will outline a possible approach that might serve as a starting point for possible future work.

The implementation described in previous chapters cannot handle yields in exception handling blocks itself because CIL does allow neither jumping into an exception handling block nor returning from it[11]. And the core principle of our approach is transforming yields into jump target labels and return statements.

We also cannot take a direct inspiration from Roslyn. While C# does not forbid *yield*s in exception handling blocks completely, it allows them only in a very limited way that is not really useful to us. It would theoretically be possible to take a look at how Roslyn handles *await*s in said blocks, because methods with *await*s are also lowered to state machines [Bezalel, 2013], but that approach was deemed unnecessarily complex for our needs. However, anyone considering actually implementing our proposed solution should use it as a resource.

5.4.1 Yields and exception handling blocks in PHP

Before going into details, we need to define what *yield*s in individual exception handling blocks even mean. A *yield* in any of these blocks, be it *try*, *catch*, or *finally*, has the same basics semantics. It returns from the function and serves as a possible continuation point for whenever the generator's *next* method is invoked again. Despite that, there are certain situations in which the *yield* behaves abnormally.

When there is a *try* block but no appropriate *catch* block and an exception is thrown, the exception is saved and does not actually truly appear until after a potential *finally* block associated with the *try* block finishes. And since the possible finally block can contain *yield*s itself, it is possible for an exception to actually appear and crawl out of the generator's *next* method only after several advancements of the generator.

There is also a special behaviour associated with *finally* blocks. When an execution of a generator stops in a block that has an unfinished *finally* block associated, be in in a *try*, *catch*, of the *finally* block itself, and the generator is about to be deleted by the GC, the *finally* block still gets run. At that point, however, it cannot contain any further *yield*s. If it does, a runtime error gets thrown.

5.4.2 Solution in Peachpie

The core idea of our proposed approach is, that we can effectively emulate the exception handling process without having to worry about the aforementioned limitations of the native one. This has some obvious drawbacks. Most importantly, it introduces a non-trivial performance hit for both a situation when an exception is thrown but also for when it is not. Depending on implementation details, the hit can either be a bit smaller or bigger but it will always be there when trying to emulate the process without using native primitives provided by the CLR.

The emulation is based on three principles. The generator instance holds the information in which exception handling block the execution currently is. Both

[11]Chapter 2.1.2

starts and ends of all exception handling blocks get transformed into specific series of statements that do two main things: keep the aforementioned information updated and they serve as labels for potential jumps. Last but not least, the whole transformed *next* method is wrapped in one giant *try* block with a *catch* in the end. A *catch* block that retrieves the information regarding which exception handling block the execution was last in and acts accordingly.

Since there is only one giant *try* block, the whole already described implementation of the *next* method can be in it. Therefore, it can contain both the initial jump table and the whole generator's code including all *yield*s. That means, there do not have to be any jumps into or from exception handling blocks, only within the big one. And regarding *return*s, they can simply be replaced with the special instruction to leave a *try* block and then a *return*.

All starts and ends of exception handling blocks must do one thing, update the information about the current exception handling block. In addition to that, the starts of *catch* blocks must include a jump label, possible conditional jump to the next catch handler for filtering only the appropriate exception type, and an unsetting of a saved exception. More about the saved exception later. The starts of *finally* blocks must contain a jump label as well, and the ends should close with a throw of a saved exception, if there is any.

The one global catch block should do following. It should retrieve the exception handling block that was just being executed. If it was a *try* block or a *catch* block with associated *finally* block, it should save the exception on the generator instance, otherwise it should just rethrow. Then, it should figure out where to jump next, in essence where to continue with the execution. If the last executed block was a *try* block, it should jump either to its *catch* block label or its *finally* block label. If it was a *catch* block, then to the *finally* block label. Lastly, just before jumping it should update and correct the information about the current exception handling block, the execution is just leaving one, after all.

To support the situation in which a generator object is going to be collected and its *finally* blocks should run, the generator type can implement an explicit *IDisposable* interface. Then in the dispose method, it can gather the exception handling blocks its execution stopped in and run all of their finally blocks. An explicit *IDisposable* interface is required instead of just using CLI finalizers because of different GC semantics between the CLR and PHP [Popov, 2012, ECMA-334, 2006].

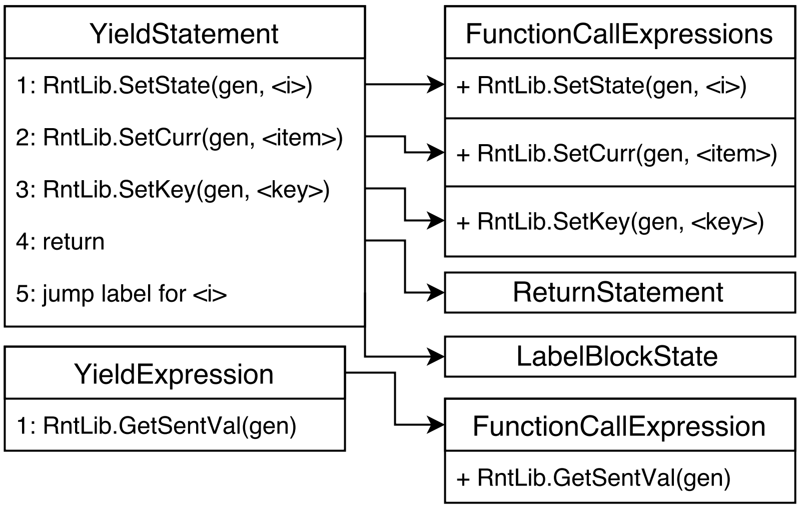
It is guaranteed that the finally blocks are triggered immediately after the last reference to the generator instance disappears in PHP. The same thing cannot be said for CLI. There is simply no way to get notified immediately after the last reference to an object disappears. Also, having arbitrary code in finalizers is highly discouraged. Since they can run in an arbitrary order it is possible that objects the generator instance might be pointing to, and that might be used by the code in the finally blocks, could have already been collected and their references set to null by the time generator's finalizer runs.

The idea outlined above is in no way comprehensive and misses a number of important details that would need to be ironed out before implementing it. In spite of that, it should serve as a good starting point.

5.5 Future work

Even though this thesis comes with a full featured implementation, there is obviously a room for further expansion. And since the project is, as of writing this thesis, still in a rapid development, it is not only possible but very probable that many of these possible follow ups will get implemented soon. That, however, does not mean they should not be noted here.

In addition to implementing the aforementioned yields in exception handling blocks, the other largest opportunity for improvement lies in employing lowering more throughout the compiler. Both the yield statement and the yield expression could be largely replaced by a lowering into lower level language constructs (Figure 5.16). While having standalone nodes was very useful for prototyping and debugging reasons, using already existing means, the rest of the infrastructure understand, is better in the long run.



YieldStatement
1: RntLib.SetState(gen, <i>)
2: RntLib.SetCurr(gen, <item>)
3: RntLib.SetKey(gen, <key>)
4: return
5: jump label for <i>
YieldExpression
1: RntLib.GetSentVal(gen)
FunctionCallExpressions
+ RntLib.SetState(gen, <i>)
+ RntLib.SetCurr(gen, <item>)
+ RntLib.SetKey(gen, <key>)
ReturnStatement
LabelBlockState
FunctionCallExpression
+ RntLib.GetSentVal(gen)

Figure 5.16: Possible lowering solution to a yield expression and a yield statement.

Lowering could also be used instead of having a custom path in the emit method of a routine symbol for generators[12]. For generator methods, the semantic binder could simply do two things. First, construct a semantic graph that would represent a generator's instantiation, initialization, and return and then use the graph as the body of the actual generator method. Second, synthesize a separate routine symbol for the generator's next method and bind the original generator method's body for it.

While not particularly interesting or complex, a *yield from* statement could also be implemented. It provides a way to specify that at some point a generator should yield elements from some iterator until it exhausts it and only then move on to the next *yield*.

There is also great potential in expanding the available code diagnostics. Peachpie compiler could, for example, warn about using *yields* in finally block, short-circuit evaluated branches, or other potentially problematic places.

[12]Chapter 5.1.8

Conclusion

In previous four sections we have first described the fundamental concepts required for understanding this thesis, then designed an algorithm to support our feature, provided an overview of said algorithm's implementation, and, in the end, proposed possible expansions.

While the work on generators support within the Peachpie compiler is by no means done, the shipped implementation provides a good foundation that can stand on its own. It brings support for all generator's features, except for yields in exception handling blocks. And while that is an useful feature, it is a more of an extension of generators than its fundamental building block. Other than that our implementation mimics the reference semantics faithfully, while expanding upon the featureset usual in other CLI based languages such as in C#.

The goal of using as much existing architecture as possible and not creating unnecessary abstractions just for generators was also achieved. While there is still room for an improvement, all generators specific code is either cleanly separated or abstracted to be used by other compiler components as well. Lastly, while not an explicitly stated goal, the compilation of generators is efficient. It does not introduce any new semantic tree or syntax tree traversals and only slightly increases the memory required for the binding phase. Due to the separation of all specific logic to a special binder, it has absolutely no impact on binding, and thus compiling, non-generator methods.

In conclusion, this thesis and the attached implementation fulfill all goals set by both the thesis assignment and us in the introduction section. On top of that, it brings a self-contained functionality to a popular open source project.

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[13].

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[14], 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.

[13] github.com/peachpiecompiler/peachpie
[14] microsoft.com/net/download/core

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

Please do note, that an assembly compiled this way will require Peachpie runtime libraries to run. These 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[15]. 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 the peachpie blog[16].

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.

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

Bibliography

Amid Bezalel. Async Await and the Generated StateMachine, January 2013. URL `https://www.codeproject.com/Articles/535635/Async-Await-and-the-Generated-StateMachine`. Accessed: 2017-07-02.

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

ECMA-334. C# language specification, June 2006. URL `https://www.ecma.ch/publications/standards/Ecma-334.htm`. 4th edition.

ECMA-335. Common Language Infrastructure (CLI), June 2012. URL `https://www.ecma.ch/publications/standards/Ecma-335.htm`. 6th edition.

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 Generator class, a. URL `http://php.net/manual/en/class.generator.php`. Accessed: 2017-06-25.

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

Nikita Popov. RFC: Generators, June 2012. URL `https://wiki.php.net/rfc/generators`. 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.