Macquarie University
Department of Computing

PROJECT REPORT

Static Type Analysis of a Dynamically Typed Programming Language

Author: Stepan Sindelar
Student ID: 43600220
Supervisor of the project: Matthew Roberts
Study programme: exchange student

Contents

1. Introduction

1.1 The Problem

Three out of the top ten programming languages in TIOBE index[1] are dynamically typed languages. One of the reasons for their popularity is that they can be easier to use and suitable for fast prototyping. But at the same time the possibility to omit type information, which might be helpful during the early stages of a software project, can lead to more error prone code, and eventually to problems in later phases of the development and maintenance. Dynamic typing is also challenging for the compiler's or interpreter's designers. With the type information, a compiler is usually able to emit more efficient code.

Programmers are aware of the possible problems with the maintenance of dynamically typed code and they often include the type information in documentation comments. However, the correspondence of the documentation and the actual code is not checked, and moreover the compiler usually does not take any advantage of having type hints in the comments.

The PHP programming language is one of the mentioned popular dynamic languages and Phalanger [2] is an implementation of a PHP compiler that compiles PHP code into the .NET intermediate code. Phalanger was developed at the Department of Software Engineering of the Charles University in Prague. A part of the Phalanger project is also an implementation of PHP tools for Visual Studio.

Because of its dynamic nature, PHP code is more difficult to analyse than code written in a statically typed language, especially if we want the analysis to be reasonably fast so that it can be used in everyday development. There is ongoing research of the static analysis methods for many different families of programming languages, including dynamic languages. The problem this project is addressing is to adapt and apply those methods on a real world and widely used programming language PHP. The result is a library that is capable of performing static analysis of PHP code and can be integrated into the Phalanger project. The library should be capable of supporting any kind of analysis, for example constant propagation. However, the main goal is to provide a type analysis in order to discover possible type related errors and mismatches with the type information in the documentation comments. Furthermore, in the future the library can be integrated into the Phalanger as a middle-end to provide optimizations for the compiler.

This project and the library has a code name *Control Flow for Phalanger* and we refer to it using this name in the following text.

1.2 Thesis structure

The following chapter describes the challenges connected with analysing source code written in the PHP programming language, approaches to static analysis of source code in general and our adoption of those to PHP. In chapter 3, we

briefly discuss existing software of this kind. Chapter 4 provides more detailed description of our implementation. The analysis has been evaluated on several middle to large sized open source PHP projects and the results are presented in chapter 5.

2. Analysing PHP Code

The PHP programming language first appeared in 1995[3]. Over the years the language has evolved and so have the ways how programmers use it. This project focuses on PHP version 5.5[1] and the aim for the analysis is to work well on PHP source code written in an object oriented style, using modern PHP patterns and idioms that are described later in this text. The analysis, however, should provide correct results for any valid PHP code of any PHP version. We do not focus only on websites, but also on PHP libraries and frameworks that by themselves do not contain any PHP files that produce HTML or any other output for the user.

2.1 PHP semantics

This section describes some important parts of the semantics of the PHP programming language, especially those that represent a challenge for static analysis.

In PHP, local or global variables, object fields and function or method parameters are dynamically typed, which means that they can hold values of completely different types at different times of the execution.

2.1.1 Local Variables

Local variables in PHP do not need to be declared explicitly. Instead the first usage of a variable is also its declaration. If a variable's value is used before the variable got any value assigned, then the interpreter generates a notice, however the execution continues and value `null` is used instead. A variable can get a value assigned to it when it appears on a left hand side of an assignment or when a reference to that variable is created, in which case it gets value `null`, but no notice is generated. References are discussed in one of the following subsections.

The scope of a local variable is always its parent function not the parent code block as in other languages like C or Java. So in the following example, the usage of variable `$y` at the end of the function can generate uninitialized variable notice, however, if `$x` was equal to `3`, `$y` will have a value although it was declared in the nested code block.

```
function foo($x) {
  if ($x == 3) { $y = 2; }
  echo $y; // uninitialized variable if x != 3
}
```

2.1.2 Global and Local Scope

PHP distinguishes two scopes for variables: global scope and local scope. Local scope is a scope of local variables within a user defined routine. Variables that are declared in global scope, that is outside of a user defined routine, are available anywhere in global scope and are called global variables. Global variables are also

[1] From this point, if the PHP version is not stated explicitly, it is implicitly 5.5.

available in user defined routines as long as they are imported into the routine's scope using the keyword `global`.

```
$g1 = 1;  // global variables g1 and g2
$g2 = 2;
function foo() {
  global $g1;
  echo $g1;   // prints the value of global variable g1
  $g2 = 4;   // sets the value of local variable g2,
  // because global variable g2 was not imported,
  // its value does not change
}
```

Global variables can be imported and used in any user defined routine. This means that even if we know some type information about a global variable's value at some point in the analysed code (e.g. straight after assignment to that variable), any time another user defined routine is invoked, we have to take into account that the other routine can change the value of the global variable even if we do not pass the global variable to the invoked routine as an argument passed by reference.

2.1.3 Closures

PHP also supports anonymous functions. An anonymous function has its own scope as any other function and its local variables are not visible to the scope where it was declared. Variables from the parent scope are available in the closure scope only if they are explicitly imported to its scope and they can be captured by value or by reference. Only the later represents a challenge for the analysis, because any code that can access the closure can invoke it and thus change the values of variables imported to the closure's scope by reference. By invoking a closure, we can influence the values of variables in a completely different and otherwise inaccessible scope.

2.1.4 References

References in PHP are similar, but not same, as pointers in the C programming language. PHP has a special operator `=&` (assign by reference) that turns the variable on the left hand side into a reference to the variable on right hand side. For example `$a=&$b`, after this, any assignment to `$a` will in fact change the value of `$b` and wherever the value of `$a` is used (e.g. in an expression), the value of `$b` is used instead.

The variable where the reference is pointing to is determined in a transitive fashion, which means that if we assign by reference another reference, the new reference will point to where the other reference was pointing to, but the intermediate link is lost. The following example illustrates this.

```
$a =& $b;       // a points to b
$c =& $a;       // c points to where a points, that is b
$a =& $d;       // a points to d, but c still points to b
```

2.1.5 Arrays

Arrays in PHP do not have to be homogenous and they can be indexed by either integers or strings. In fact, PHP arrays are hash maps rather than arrays in the usual sense and that is also how they are implemented internally.

String indexed heterogenous arrays are often used as flexible ad-hoc structured data type. Instead of defining a class with required fields, one can use what would be a field name as an index into an array. Such arrays are usually indexed only with finite number of constant string values.

In this light, it is no surprise that using the subscribe operator `[]` with string index on an object instance will access the field with the same name as the index value.

2.1.6 Interesting Control Flow Structures

The `break` and `continue` statements with optional numeric argument are supported in PHP in a similar way as in other standard imperative programming languages. There are, however, important differences to be noted.

Firstly, The numeric argument can be an arbitrary expression in some of the older versions of PHP, in which case we cannot statically determine the target of the jump for the control flow resolution.

Secondly, the `switch` statement is considered a loop for the purposes of both `break` and `continue`. The semantics of `break` is intuitive. One of the meaningful use cases is to break a loop from within a `switch` by using `break 2`. The semantics of `continue` statement is perhaps not so intuitive: within a `switch` it works the same way as `break`.

Switch Statement Semantics. The basic semantics of the `switch` statement in PHP is again very similar to that of other standard imperative programming languages. The `switch` statement in PHP permits an arbitrary expression as the value to be used for comparison with values of its `case` labels. Furthermore, the values of `case` labels can also be arbitrary expressions and because we are in a context of dynamic programming language, they can again evaluate to a value of any type.

At runtime, the switch expression is evaluated only once at the beginning, and if it has an undefined value (undefined variable, void function call), then the control flow goes directly to the default item, without evaluating the expressions in the case items. If the value is defined, then it is one by one compared to the values that the `case` labels evaluate to. If a `case` label evaluates to `boolean` value, then it is used to decide whether to jump to that `case` item or continue with evaluating the value of the next `case` label. Note that the value of switch expression is not compared to the `case` label value. If a `case` label evaluates to a complex type (`object` or `array`), it is ignored and evaluation continues with the next `case` label. And finally, if a `case` label evaluates to an `integer`, `float` or `string` value, it is compared to the switch expression. All these expressions can have side effects due to usage of assignments as expressions or calls of functions with side effects.

PHP also allows to place the `default` label anywhere in between the other `case` labels. This can be used for fall-back to or from a `case` item as in the

following code sample that is abbreviated version of actual code taken from the WordPress[4] code base.

```
switch ( $status ) {
    default:
    case 'install':
        $actions[] = '<a class="install-now" ...';
        break;
    case 'update_available':
        $actions[] = '<a class="install-now" ...';
        break;
    case 'newer_installed':
    case 'latest_installed':
        $actions[] = '<span class="install-now" ...';
        break;
}
```

2.1.7 Conditional Declarations

User defined functions, classes, etc. are declared in a global scope in PHP, that is a scope where one can as well place any arbitrary code. Therefore a declaration can be wrapped in any control structure. It is not allowed to redeclare once declared symbol, however.

A typical use case is to dynamically import a file that may provide some functions and then check, using `function_exists`, whether the functions were indeed declared and if not, provide default implementation. This is a pre-object-oriented way of doing overriding and is usually not to be found in modern projects. Nonetheless, WordPress still relies on this pattern in parts of its code base.

Although the mentioned pattern could be deemed as reasonable and useful. This feature allows to write very problematic code as in the following example that may or may not crash on fatal errors "Cannot redeclare foo()" or "Call to undefined function foo()" depending upon the user input.

```
while ($_POST['a'] != 3) {
   function foo() { return 5; }
   $_POST['a'] = $_POST['b'];
}
echo foo();
```

2.1.8 Auto-loading

Historically, in PHP in order to reference any symbol from a different file, one had to import that file explicitly. Newer versions of PHP support customized auto-loading. A used defined routine can be invoked by the runtime every time an undefined class is referenced. The auto-loading routine is then responsible for importing the file(s) that contain the code of the required class.

Auto-loading routine can use arbitrary logic to determine what file(s) to import, in fact, it can execute arbitrary code. Typical pattern used for example in Zend Framework[5] before namespaces were introduced to PHP is to have a file

per class and use class names in form of `CodeFolder1_SubFolderName_FileName` for a class placed in file `CodeFolder1\SubFolderName\FileName.php`.

2.1.9 PHPDoc Annotations

Although not part of the official PHP syntax, there is a widely recognized format for documentation comments of JavaDoc style called PHPDoc. PHPDoc comments may contain type information that cannot be expressed using PHP syntax. For example, PHP allows "type hints" for routine parameters, but only for class types, not for primitive types like `int`. However, primitive type expectations can be included in the documentation comments. The important difference is that PHP will throw an exception at runtime if a routine is invoked with a parameter of different type than what its "type hint" is. The documentation comments, on the other hand, are of course ignored by the runtime.

The PHPDoc defines a fairly advanced syntax for expressing type information. It supports multiple primitive and class types, homogenous and heterogenous arrays as well multidimensional arrays, and some constants like `false`. For example, in the following code the documentation comment tells us that function `foo` can return either `null`, or `false` (but should never return `true`), or an array of `integer` values.

```
/**
 * @return null|false|int[]
 */
function foo() { ...
```

2.2 Static Code Analysis

Static analysis of source code is an analysis that is performed without executing the code. This means that we do not need to have a web server, for example, in order to analyse code of a web application. We can also guarantee some properties of the analysis that would not be possible to guarantee if we executed the code. Namely the halting property and upper bounds on time and space complexity. Arbitrary code may not halt if executed, but static analysis of such code can still halt and give results.

Static analysis can be used to get possible types of an expression in a dynamically typed language, to find out expressions that have constant value through constant propagation and many other problems. Static analyses usually do not give accurate solution, but its approximation, which can be an over-approximation or an under-approximation and it is up to the designer and user of the analysis which one is acceptable for his or her[2] purposes.

2.2.1 Terminology

Static analyses are usually described in the form of inference rules. An example of an inference rule can be "if the types of expressions e_1 and e_2 are integers,

[2] "His" or "he" should be read as "his or her" or "he or she" through the rest of the text.

then the type of expression $e_1 + e_2$ is integer". Those rules can be more formally described with the following notation

$$\frac{\vdash e_1 : Integer \quad \vdash e_2 : Integer}{\vdash e_1 + e_2 : Integer}$$

where above the horizontal line we have hypothesis and below is the conclusion. The exact notation is not important, we will be using it intuitively to illustrate the ideas that we describe.

Flow Sensitivity. The example inference rule is not valid, if we admit that evaluation of expression e_1 can influence the type of expression e_2 or vice versa. In such case, the ordering of the expressions is important, but not captured in the inference rule. Therefore this rule is *flow insensitive*. If we make the hypotheses more complex using additional constraints to capture the ordering and dependency of the expressions, it will be *flow sensitive*.

Path Sensitivity. If we admit conditional control flow statements, like if-then-else, we can have more possible paths through the program. In our example, let us say that there is an if statement before the expression e_1 and the expression e_1 can evaluate to different type if the then branch of the if-then-else statement is taken than if the else branch is taken. If the inference rules do not model this, like our example rule, we say that the inference system is *path insensitive*, otherwise we say that it is *path sensitive*.

TODO: I should do some more reading about path sensitivity, is this a correct description? Add an example of how such inference rule would look like?

Abstraction. Another example of problem that can be partly solved with static analysis is the sign of integral variables. We can have inference rules of the following form.

$$\frac{\vdash v_1 : -10(sign : \ominus) \quad \vdash v_2 : 3(sign : \oplus)}{\vdash v_1 + v_2 : -7(sign : \ominus)}$$

However, the implementation would not be very efficient and we sometimes do not have the full information about variables values, but in some cases we can deduce another less precise, but still useful piece of information by other means. For example, variable of type `unsigned integer` will always be positive, we can count on that even if we do not know the actual value. What we can do is to abstract the possible integral values with set $\{0, \ominus, \oplus\}$ with the following meanings

- $\ominus$ represents all negative integers,
- $\oplus$ represents all positive integers,
- 0 represents zero,

and rewrite the inference rules as follows:

$$\frac{\vdash v_1 : \ominus \quad \vdash v_2 : \ominus}{\vdash v_1 + v_2 : \ominus}$$

Nonetheless, there is another problem. What to do when we have $\ominus$ and $\oplus$ in the hypothesis.

$$\frac{\vdash v_1 : \ominus \quad \vdash v_2 : \oplus}{\vdash v_1 + v_2 :?}$$

The solution is to extend the domain so that it is closed under all operations. We add another element to our set:

- $\top$ represents an unknown value (either positive, negative, or zero).

Then the rule will be:

$$\frac{\vdash v_1 : \ominus \quad \vdash v_2 : \oplus}{\vdash v_1 + v_2 : \top}$$

And for example another rule dealing with $\top$ in hypothesis:

$$\frac{\vdash v_1 : \ominus \quad \vdash v_2 : \top}{\vdash v_1 + v_2 : \top}$$

2.2.2 Data Flow Analysis

Data Flow Analysis (DFA) is a static analysis framework for compiler optimizations and verification that scales to large code bases [6], [7].

Control Flow Graph. DFA is typically performed on a control flow graph, although there are exist approaches to DFA without explicit control flow graph construction [8].

Control flow graph nodes, also called basic blocks, are program statements that are always sequentially executed. Directed edges represent the control flow between basic blocks, for example, jumps in the control flow due to conditionals, `goto` statements or any other statements that can change the flow of the program. Control flow graphs usually contain two special nodes: entry node and exit node. The entry node does not have any incoming edges and all the paths lead to the exit node. An example of a control flow graph is given in figure 2.1.

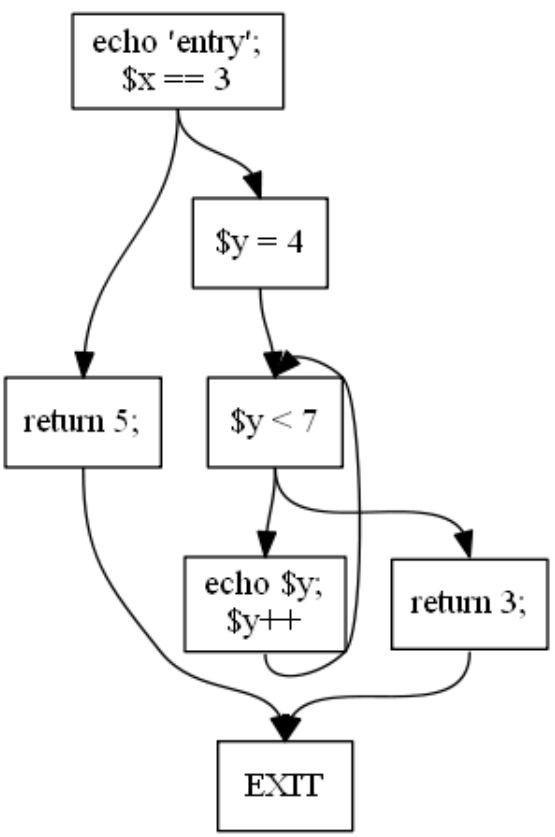
Lattices. The reasoning behind correctness and termination of DFA is based on an algebraic structures called lattices. A lattice is a partially ordered set in which every two elements have a least upper bound, called supremum, and a greatest lower bound, called infimum.

Bounded lattice is a lattice that has a greatest element and a least element, usually denoted as $\top$ and $\bot$. A bounded lattice is depicted in figure 2.1.

What is important about lattices for DFA analysis is that if we have a bounded lattice $(S, \leq_s)$ and a function $f : S \to S$ that is monotonous, i.e. $\forall a \in S : a \leq_s f(a)$, then $\forall x \in S\ \exists k \in \mathbb{N}$, such that $f^k(x) = f^{(k+1)}(x)$. $f^k(x)$ is called a fixpoint.

Intuitively, f has to have a fixpoint because for every argument y, it must either return y itself, but then y is a fixpoint, or it returns another element that is strictly greater than y, but this cannot go on forever, because eventually f will be given $\top$ for which it does not have any other option but returning $\top$ and we have a fixpoint again.

The following few paragraphs will finish the description of Data Flow Analysis.




```php
echo 'entry';
if ($x == 3)
    $y = 4;
else
    return 5;

while ($y < 7) {
    echo $y;
    $y++;
}

return 3;
```

Table 2.1: Control flow graph

2.2.3 Intraprocedural Analysis

So far we have been discussing an analysis of a single function or method[3]. However, if we want to analyse whole program or a library, the interaction between the routines can be taken into account to make it more precise.

Context Sensitive Intraprocedural Analysis. The most precise solution would be to take into account the calling context when analysing a function. In different contexts, the function can be, for example, given different parameters values which may then influence the result of the analysis. More precise result for a specific call site context can be propagated to that call site, yielding another gain in precision when analysing the function that realises the call.

The following few paragraphs will discuss feasibility of Context Sensitive Intraprocedural Analysis, because call sites are not always known, practical consequences and usual approaches to make Context Sensitive Intraprocedural Analysis

[3]We will use term routine to designate a global function, static or instance method

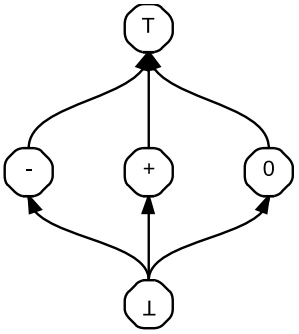


Figure 2.1: Bounded lattice with 5 elements.

more scalable.

2.3 Control Flow for Phalanger Approach

Description of the analysis used in our case using the terminology built up in the previous section. It will be more abstract description, without technical details about implementation.

3. Existing Software

3.1 Phantm[9]

3.2 HipHop Type Inference for Hack

3.3 Weverca: Web Verification Tool[10]

4. Implementation

4.1 Implementation Specific Constraints

One of the requirements was that the project should be implemented in the context of the Phalanger project. It should be ready to be plugged in between the Phalanger's front-end and back-end and it should also provide public interface useful for the Phalanger PHP Visual Studio tools.

Abstract Syntax Tree. Phalanger front-end parses PHP code into an Abstract Syntax Tree (AST) [7] structure. This structure is then traversed by the back-end using the visitor design pattern [11]. Phalanger does not use any other intermediate representation than AST and the back end transforms the AST directly to Microsoft Intermediate Language (MSIL).

In order to reduce the memory consumption and provide better modularity, Phalanger code went through small architecture refactoring before this project was started. The classes representing the AST nodes originally contained the code and data needed for emitting the corresponding MSIL opcodes. This, however, represents a coupling between the front-end and back-end and the front-end cannot be used just on its own. In the new version, the classes representing AST nodes are capable of storing additional attributes in an extensible way and the back-end have been rewritten to be an ordinary AST visitor that uses the extensible AST nodes attributes. Additional attributes provide a way to annotate AST nodes with any additional information, which in our case will be the results of the analyses.

Integrated Development Environment Integration. The PHP Tools for Visual Studio use Phalanger front-end in order to parse PHP code into AST and then the AST is again traversed to provide code completion and other features. All the AST nodes hold necessary pieces of information, e.g. the position in the source file or documentation comments.

The aim of this project is to provide the results of type analysis and other analyses to the integrated development environment so that the possible errors and warnings can be visualised, e.g. underlined.

The longer term aim of this project, not in the scope of this thesis, is to replace the existing algorithms for code completion, "jump to definition" and "find usages" features. Because with a dynamic language like PHP, it is not trivial to find all the usages of, for example, a class or determine a definition of, for instance, a field accessed on some local variable. In order to provide more precise results, type analysis is needed.

One of the challenging parts of integrated development environment integration is also dealing with incomplete code that is being typed in by the user. Therefore one of the requirements was also that the analysis should be capable of performing an ad-hoc re-analysis of once analysed code with a new statement added. This ad-hoc re-analysis should be, if possible, more effective that doing the whole analysis again.

4.2 Overall Design

The project is divided into several modules.

- Control Flow Graph,
- Intermediate Representation of PHP Code (Phil, RPhil),
- Generic Data Flow Analysis Framework,
- Tables with Type and Other Information,
- Concrete Analyses:
  - Dead Code Elimination,
  - Aliasing Analysis,
  - Constant Propagation,
  - Type Analysis.

The interactions between those modules on a conceptual level when performing an analysis are depicted in diagram 4.1. Green elements represent extension points. Red arrows represent the core flow of the algorithm.

Intermediate Language. One of the goals of the design was to stay as close as possible to the original AST representation, so that the results of the analysis can be easily propagated to an IDE and that the Phalanger back-end does not have to be rewritten in order to leverage the this project.

Control Flow for Phalanger uses two intermediate representations on a conceptual level, however, they are usually not explicitly constructed as described later in the text. The purpose of the intermediate representations is to simplify the design of the analyses.

Phil stands for *PHP Intermediate Language* and is very abstract representations close to the original AST. Phil contains only statements and expressions that can be in a Basic Block, therefore it does not contain most of the control flow changing statements like if, switch, or loops. A Phil statement represents the smallest single step of an execution that can change state of variables or global state or throw an exception. Syntactic constructs like `$i++` are unfolded to `$i = $i + 1`, which is then split to evaluation of binary expression and assignment expression that uses the result of the binary expression. In some sense, Phil can be viewed as a three address code.

RPhil stands for *Resolved PHP Intermediate Language*. RPhil is basically a Phil with resolved symbols where possible. By resolved symbols, we mean references to the elements of Type Tables discussed in one of the following paragraphs. In order to resolve the symbols, the module building RPhil can use names explicitly expressed in the code, for example, for direct local variable access `$a`, or the results of an analysis, for instance, the results of type analysis to resolve method calls and fields references. RPhil itself is typically consumed by the analyses, so the accuracy of RPhil and subsequently of the analyses results can be improved by iterative execution of the analyses.

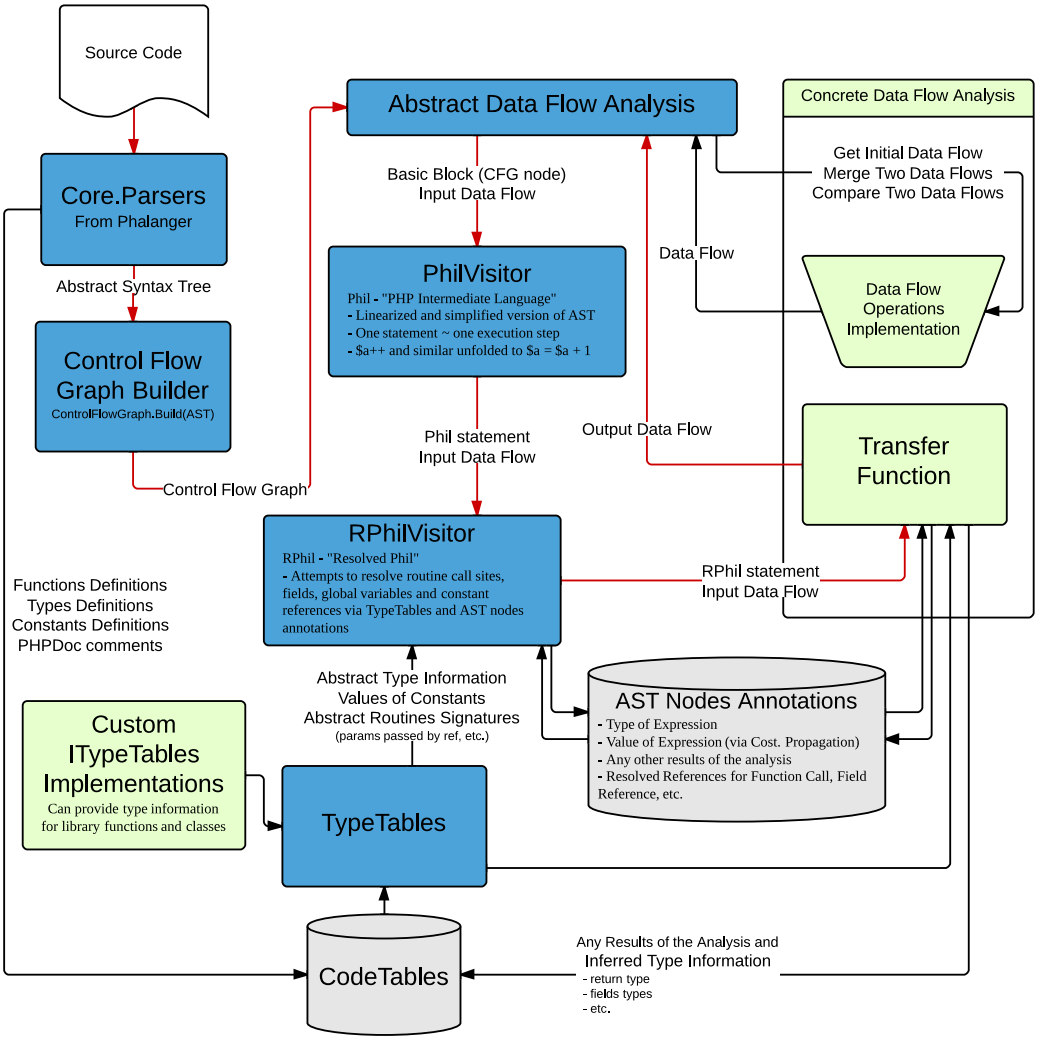


Figure 4.1: Control Flow for Phalanger Design Overview

Analyses. The Data Flow Analysis is performed on Control Flow Graph nodes called basic blocks. Each basic block contains a list of Abstract Syntax Tree elements. This list is then, usually on the fly, transformed to corresponding PHP Intermediate Language elements before its elements are given to the flow function of a concrete Data Flow Analysis implementation.

Basic block can also contain already transformed PHP Intermediate Language elements. However, the interface does not change from the point of view of a concrete Data Flow Analysis implementation.

The Control Flow for Phalanger also contains and allows to plug-in simple analyses that are performed for all basic blocks sequentially without taking the control flow into account. Each basic block is then analysed only once. An example of such analysis is Aliasing Analysis.

The last analysis type, which stands aside, is Dead Code Elimination. It is performed on the Control Flow Graph, but traverses it on its own as opposed to a concrete Data Flow Analysis that only visits basic blocks in order determined by the Data Flow Framework, i.e. does not perform the graph traversal on its

own. Control Flow for Phalanger also allows to add custom analyses that are performed on the raw Control Flow Graph.

Analyses Results. The results of an analysis can be annotations added to the AST node objects, or annotations of basic blocks, which support extensible attributes in the same way as AST nodes.

The results of an analysis can also be pushed into the Code Tables. Code Tables gather relevant information about code elements like classes, functions, etc. It is mainly type related information, for example, a return type of a function, but also which parameters are passed by reference or whether the function returns a reference.

Type Tables. This module provides almost the same information about routines, types, global variables and constants as Code Tables. However, in this case the information is abstract, because it might not be directly related to an actual element that can be found in the source code. This is the case of library functions and classes, but it also gives a possibility to merge the information of two or more conditionally declared elements.

Although this kind of support for conditional declarations has not been implemented, the design is prepared for it, shall it be considered an important feature in the future. At the moment, if two or more declarations are found, the Type Tables behave as if they did not know the element at all, which preserves correctness for the price of loosing precision.

Extensibility. The whole project is designed as a class library and framework with many extension points. Some of the functionality can be used independently. For example, Control Flow Graph builder to generate diagrams.

Nonetheless, in order to provide better usability, Control Flow for Phalanger also contains a Facade class `AnalysisDriver` that plugs in together all the necessary objects and provides a simple interface to perform a defined analysis of a file or a given piece of code.

4.3 The Intermediate Language

The aim of the intermediate representation is... The three address code can be built by traversing the AST in the order of execution... Some AST elements represent more than one execution step...

RPhil: the symbols are resolved using defined AST nodes annotations for constant values and type information... `VarLikeConstructInfo` abstracts variables accesses...

4.4 Control Flow Graph

Description of the Control Flow graph construction.

4.5 Data Flow Analysis

Architecture of the generic algorithm implementation using generics in C# in a way that the graph the analysis can be performed upon an arbitrary graph (not only CFG), which can be useful if in the future some of the analyses will be performed, for example, on a definition-use graph.

4.6 Tables

Intra-procedural dependencies handling `DependencyResolver`.

4.7 Analyses

4.7.1 Dead Code Elimination

4.7.2 Aliasing Analysis

4.7.3 Constant Propagation

4.8 Type Analysis

- *Type Information representation*
- *Data Flow representation*
- *AST annotations*

4.9 The Analyser pipeline and AnalysisDriver

Description of the high level public interface and the phases of the full analysis process.

5. Results

The project has been evaluated on the following open source PHP frameworks and websites.

- PHPUnit: a port of JUnit unit testing framework for PHP.
- Zend Framework: popular general purpose PHP framework.
- Nette: another popular PHP framework for building websites.
- WordPress: on of the most popular content management systems.

An evaluation always started with downloading a git repository with the latest source code of given PHP project. Than the analysis was run and all the discovered problems were collected and categorized. Actual errors were rectified and recorded as commits in the git repository.

5.1 Problems Taxonomy

The problems were divided into few main categories described below. Some of the problems recurring across all the projects are discussed in this section, while the more project specific problems are investigated later each project's section.

Style: a category of problems that are not directly affecting the functionality of the application, but they can be considered as a bad coding style.

> *Relaying on default return value* – when an execution of a routine does not end with a return statement, its return value is `NULL` by default, which can then be cast to values of other types, like `false` for example. Developers rely on this feature and omit the return statement if they want to return `NULL` or something that `NULL` can be cast to.

Documentation: inconsistency of the PHPDoc type documentation and what the the code does. This category includes only cases where it is clear that the documentation is wrong, not the code, for example, due to updates in the code that were not reflected in the documentation. The inconsistency may also indicate another problem, in which case it does not belong to this category.

Interestingly, most of the inconsistencies of type of a parameter of a function call typically lead through several routines that only forward the parameter to the next routine until eventually the routine that has a wrong documentation is reached.

> *Missing* `false` *in return value type documentation* – this is common pattern in PHP where a routine returns `false` when it fails to do what it was supposed to do. For example, function `fopen` returns `resource` of `false` if the resource could not be opened. It is so common that developers tend to forget to put `false` into the documentation.

Actual Error: includes all problems that can cause an unexpected exception or unexpected runtime error or notice.

False Positive: problems reported by the tool that are not in fact real problems. Includes only false positives that cannot be eliminated because of fundamental design reasons or are not intended to be eliminated, because it would cause some actual positives to be missed.

Unused routine arguments – when a method is an override of some base method, it can have the same signature and if some of the parameters are not used, they are not reported. There are however some cases where the routine is implementing some interface by convention that is not explicitly expressed in the syntax of PHP. For example, the pre-object-oriented pattern for global functions overriding. In such case the analyzer cannot determine that the unused parameter is in fact a part of an interface. Note that such function could omit the unused parameter and everything would work the same, therefore this may or may not be considered a false positive.

Amendable false positive – false positives that are reported, although the algorithm the analyzer is using should not report them. Those indicate errors in the implementation.

Built-in documentation errors – false positives due to the inaccuracy of the documentation of built-in functions and classes that was used in the experiment.

5.2 Summary

Category	PHPUnit	Zend Framework	Nette	WordPress	total
Style	6	NA	NA	NA	6
default return value	2	NA	NA	NA	2
Documentation	10	NA	NA	NA	10
missing false	3	NA	NA	NA	3

5.3 PhpUnit

PHPUnit is a mature and well established project that has been developed for more than 6 years by 156 contributors. Being a unit testing framework, PHPUnit itself has extensive unit test suite. For the experiment the master branch of the clone of the repository retrieved on 18.5.2014 was used.

5.3.1 Individual Errors.

Q: move the table to an Appendix? Should I do such table for each project? – lots of data and work to be done. Maybe only for PHPUnit and for other projects only discuss interesting errors and provide quantitative data in the summary table above.

File	Line	Category	Note
`Framework\TestCase.php`	1722	Style	Foreach used in form `foreach($array as $key=>$val)` although the `$key` variable was not used anywhere.
`Framework\TestCase.php`	1726	Style	Discussion follows.
`Framework\Assert.php`	1861	Actual Error	Discussion follows.
`Framework\Assert.php`	1960	Documentation	The routine code allows one of the arguments to be an `array` and works with it as such, but the documentation states that it can only be `boolean`.
`Framework\Assert.php`	1896	Documentation	`assertSelectEquals` method restricts its parameter type to be `integer` and is invoked with `boolean`. However, the parameter value gets only forwarded to `convertSelectToTag`, which accepts any type (mixed).
`Util\XML.php`	544	Documentation	Missing `false` in return value documentation.
`Util\Test.php`	294	Documentation	
`Util\GlobalState.php`	351	False Positive	Built-in documentation error.
`Util\Test.php`	360	False Positive	
`Framework\TestCase.php`	1941	Documentation	Field `mockObjectGenerator` is documented to have type `array`, but value of type `MockObject_Generator` is assigned to it. The documentation should be updated.
`Util\GlobalState.php`	351	False Positive	Built-in documentation error.
`Util\Test.php`	46	False Positive	Function `trait_exists` is conditionally declared if it does not exist. It follows the same signature of actual built-in function `trait_exists`, but it has empty body, therefore it does not use its arguments.

5.3.2 Actual Error When Handling `DOMElements`

The error is related to the following code (shortened).

```
function assertEqualXMLStructure(
  DOMElement $expectedElement/*, ...*/) {
  ///...
  PHPUnit_Util_XML::removeCharacterDataNodes($expectedElement);
  PHPUnit_Util_XML::removeCharacterDataNodes($actualElement);
  //...
  for ($i = 0; $i < $expectedElement->childNodes->length; $i++) {
      self::assertEqualXMLStructure(
          $expectedElement->childNodes->item($i) /*<<< error */
          /*...*/);
}
```

The method `assertEqualXMLStructure` accepts only instances of `DOMElement`, but it invokes itself recursively with first argument of type `DOMNode`. Because according to the PHP documentation the value of the `childNodes` property of interface `DOMNode` is an instance of `DOMNodeList` and the method `item(integer)` of `DOMNodeList` returns `DOMNode`, it is a type mismatch error as `DOMNode` is not subtype of `DOMElement`.

In the PHP implementation of DOM model, the only implementations of `DOMNode` either inherit from `DOMElement` or implement `DOMCharacterData`, and those are removed from the `childNodes` collection by `removeCharacterDataNodes`. Therefore, in most cases, this code behaves as expected.

However, the `DOMNode` interface can be implemented by any user defined class, which does not have to inherit from `DOMElement`, and if an instance of such class was present in the `childNodes` collection, the code would cause an exception when trying to invoke `assertEqualXMLStructure` with an argument of wrong type.

Note that if method `removeCharacterDataNodes` removed all the child nodes that are not instances of `DOMElement`, the code would be correct, but the error would still be reported, therefore it would be false positive.

5.4 Zend Framework

5.5 Nette

5.6 WordPress

6. Conslusion

6.1 Future Work

Integer interval analysis, integration with the compiler back-end.

Bibliography

[1] "Tiobe index for march 2014." http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 2014-09-03.

[2] J. Benda, T. Matousek, and L. Prosek, "Phalanger: Compiling and running php applications on the microsoft .net platform," *.NET Technologies 2006*, 2006.

[3] "Php: History of php." http://www.php.net/manual/en/history.php.php. Accessed: 2014-02-04.

[4] "Wordpress ¿ blog tool, publishing platform, and cms." http://wordpress.org/. Accessed: 2014-07-05.

[5] "Zend framework." http://framework.zend.com/. Accessed: 2014-07-05.

[6] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

[8] M. Mohnen, "A graph—free approach to data—flow analysis," in *Compiler Construction*, pp. 46–61, Springer, 2002.

[9] E. Kneuss, P. Suter, and V. Kuncak, "On using static analysis to detect type errors in php applications," tech. rep., 2010.

[10] D. Hauzar and J. Kofron, "Hunting bugs inside web applications,"

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

List of Tables

Attachments