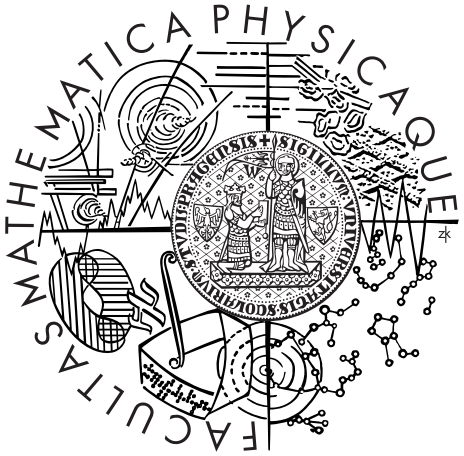
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Štěpán Šindelář

Implementing control flow resolution in dynamic language

Department of Software Engineering of the Charles University in Prague

Supervisor of the master thesis: Filip Zavoral, Ph.D.
Study programme: Software Systems
Specialization: specialization

Prague 2014

Dedication.

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

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

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

Název práce:

Autor: Bc. Štěpán Šindelář

Katedra: Katedra softwarového inženýrství Univerzity Karlovy

Vedoucí diplomové práce: Filip Zavoral, Ph.D., TODO: pracoviště

Abstrakt:

Klíčová slova: dynamické programovací jazyky, statická analýza, PHP, Phalanger, .NET

Title: Implementing control flow resolution in dynamic language

Author: Bc. Štěpán Šindelář

Department: Department of Software Engineering of the Charles University in Prague

Supervisor: Filip Zavoral, Ph.D., TODO: pracoviště

Abstract: Dynamic programming languages allow us to write code without type information. The type of variables can dynamically change during execution. Although easier to use and suitable for fast prototyping, dynamic typing can lead to error prone code, difficult maintenance and is challenging for the compilers or interpreters. Programmers often use documentation comments to provide the type information, but the correspondence of the documentation and the actual code is usually not checked by the tools. In this thesis, we focus on one of the most popular dynamic programming languages: PHP. We have developed a framework for static analysis of PHP code as a part of the Phalanger project – the PHP to .NET compiler. The framework supports any type of analysis, but in particular, we implemented type inference analysis with emphasis on discovery of possible type related errors and mismatches between documentation and the actual code. We use a modular approach to static analysis of PHP applications, which is different to most of the other tools of this kind. The implementation was evaluated on real PHP applications and discovered several real errors and documentation mismatches with a good ratio of false positives.

Keywords: dynamic programming languages, static analysis, PHP, Phalanger, .NET

Contents

1. Introduction

1.1 The Problem

Three out of the top ten programming languages in TIOBE index[1] – an indicator of programming languages popularity, 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 into 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 many kinds 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 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 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.

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

Section Static Code Analysis gives a brief overview of existing static analysis methods, especially Data Flow Analysis (DFA)[4][5], which has become a de-facto standard type of analysis for most of the optimizing compilers and other more complex static analysis methods are either directly based on DFA or on the ideas behind DFA. This section includes references to relevant literature and is not meant as a comprehensive description, but should provide a context for the following section Control Flow for Phalanger Approach, where we discuss how we used the existing techniques for the purposes of our PHP code analysis.

2.1 PHP Semantics

2.1.1 Dynamic Typing

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.

2.1.2 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.

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

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

2.1.3 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 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.

2.1.4 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.

2.1.5 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 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
```

2.1.6 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.

2.1.7 Interesting Control Flow Structures

The `break` and `continue` statements with optional numeric argument are supported in PHP in a similar way as in other 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 permits placing 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[6] 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;
}
```

2.1.8 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 permits 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();
```

2.1.9 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 user 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[7] 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`.

2.1.10 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, it is an 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 Program State

Execution of a program can be seen as a series of transformations of the program state. Each individual program instruction, when executed, can change the program state and produces its *output state*. How exactly is defined by the instruction's semantics and it typically depends on the *input state*, which is a program state produced by the previously executed instruction.

The goal of a static analysis is to devise some useful piece of information about how instruction i can change the program state, so that it can be used for program optimization or to reveal potentially problematic instructions. For example, if assignment $a = 4/b$ always assigns constant value 1 to a, because b happens to be equal to 4 in any possible program state preceding $a = 4/b$, we can change $a = 4/b$ to $a = 1$, which has the same affect to the program state. If we instead found out that b is always equal to 0, we would know that this instruction will cause an exception.

The result we expect from a static analysis is, for each instruction in the program, provide some useful property of program state transformation that always holds every time the instruction is executed. The analyses differ in the properties they compute. We will call such computed property a *data-flow*.

2.2.2 Control Flow Graph.

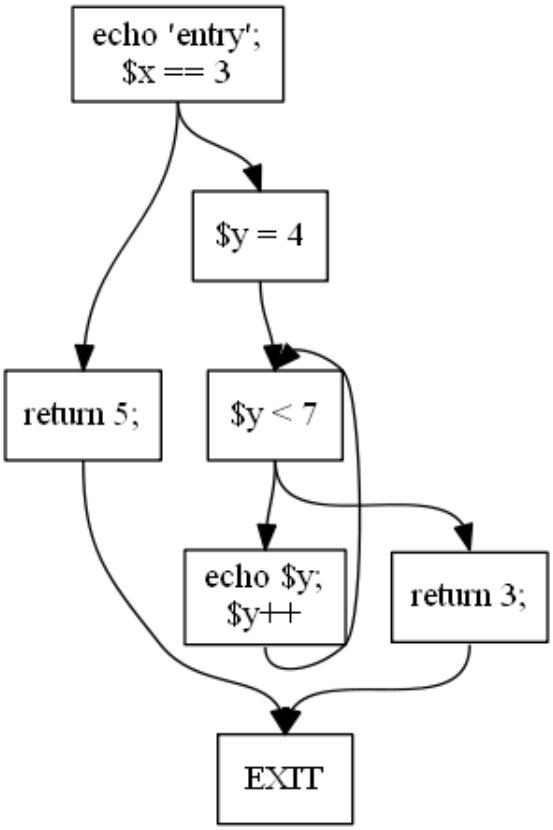
DFA is typically performed on a control flow graph, although there exist approaches to DFA without explicit control flow graph construction [8].

Control flow graph nodes, also called basic blocks, are program statements that are always executed sequentially. 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.

2.2.3 Transfer functions

We say that the input state of a statement is associated with the *program point before* the statement and the output state is associated with the *program point after* the statement. Our aim is to calculate *data-flow* value for both program points for each statement, denoted $IN(s)$ and $OUT(s)$ for a statement s.

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




```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

Single Statement

We can express the *data-flow* for *program point after* a statement as a function of the *data-flow* of *program point before* the statement, also called *transfer function*. Formally if f_t is a *transfer function*, then $f_t(IN(s)) = OUT(s)$. Each type of statement will have a different *transfer function* that will reflect the semantics of the statement.

Example: our analysis tracks type of local variables, so the *data-flow* is a map from variable names to their type. In such setting, a *transfer function* for assignment statement `$a=$b`, will take the input *data-flow* $flow_{in}$ and will return $flow_{out}$, such that $flow_{out}(x) = flow_{in}(x)$ for every variable name x, except for $flow_{out}(\$a) = flow_{in}(\$b)$. In other words, the type of all the variables stays the same, except for `$a` whose type changes to whatever is the type of `$b` is.

In practice, there is only one *transfer function* for all the assignment statements that is parametrized by the left-hand side and right-hand side of the assignment. And in general, each statement in the programming language has usually its "meta" *transfer function* that is parametrized not only by the input *data-flow*, but by the statement structure.

Transfer functions are often described in the form of inference rules. An example of an inference rule can be "if the type of variable `b` is `Integer` in state S_1, then after executing statement `a=b`, we get a new state S_2 that is the same as S_2 except variable `a` is of type `Integer` in S_2". Those rules can be more formally described with the following notation

$$\frac{S_1 \vdash b : Integer}{[a = b, S_1] \rightarrow S_2, S_2 \vdash a : Integer}$$

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.

Statement Interaction

The *data-flow* for *program point before* a statement depends on the *data-flow* of *program point after* the last executed statement. Let us consider a basic block B with statements $s_0, s_1, ..., s_n$.

In the case of any $s_i \neq s_0$, that is any statement but the first one, there is only one statement whose execution can precede the execution of s_i and that is the previous statement in the sequence: s_{i-1}. So we have $IN(s_i) = OUT(s_{i-1})$ for $i \in [1..n]$.

In fact, thanks to this property, we can define *transfer function* f_B of a basic block as composition of transfer functions of its statements. Formally:

$$f_B = f_{s_n} \circ f_{s_{(n-1)}} \circ ... f_{s_0}$$

where f_{s_i} is the transfer function for statement s_i. Furthermore, we denote *data-flow values* of basic block as $IN(B) = IN(s_1)$ and $OUT(B) = OUT(s_n)$. From the definitions we can observe simple identities:

$$f_B(IN(s_1)) = f_B(IN(B)) = OUT(B) = OUT(s_n)$$

The first statement s_1 of the basic block has to be handled differently. Its execution can be preceded by execution of the last statement of any of the basic blocks $B_1, B_2, ..., B_m$ that precede basic block B in the control flow graph. One of the possibilities to deal with this, is to combine *data-flows* $OUT(B_1), ..., OUT(B_m)$ into single *data-flow* that approximates all of them. Nonetheless, the way the *data-flows* are combined depends on the concrete *data-flow* type and thus should be defined by the analysis. We denote the function to combine *data-flows* as $MEET$.

Equations for Data-Flow

When we put everything together, we get a set of equations:

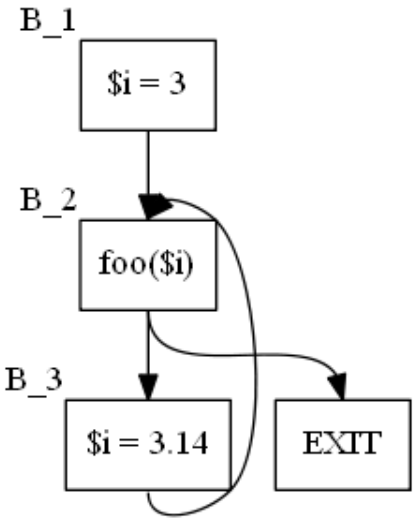
$$\begin{aligned} OUT(B_i) &= f_{B_i}(IN(B_i)) \\ IN(B_i) &= MEET(OUT(P_0), ..., OUT(P_m)) \end{aligned}$$

where $P_0, ..., P_m$ are predecessors of B_i in the control flow graph. The analysis must also define value of $IN(START)$, that is *input state* for the initial node, which does not have any predecessors.

Example

We will illustrate the system of *data-flow* equations on a simple example. The subject of our analysis will be the type of the variable `$i` in the code sample in figure 2.2.

Data-flow will be a set of possible types of variable i and initial *data-flow* of the start node will be an empty set.




```
$i = 3;
while (foo($i)) {
    $i = 3.14;
}
// exit
```

Table 2.2: Code for Data-Flow Example

Transfer functions: we will assume that a function call does not change state, therefore the transfer function of function call is identity – $OUT(s) = IN(s)$. Assignment of a constant `c` of type `T` to `$i` changes type of `$i` to `T`.

MEET operation will be union of the sets of possible types of `$i`.

The set of equations is in this case

$$
\begin{aligned}
OUT(B_1) &= f_{B_1}(IN(B_1)) = f_B(\emptyset) \quad \text{(initial node)} \\
OUT(B_2) &= f_{B_2}(MEET(OUT(B_1), OUT(B_3))) \\
OUT(B_3) &= f_{B_3}(OUT(B_2))
\end{aligned}
$$

Knowing that f_{B_1} and f_{B_3} are constant functions, because the assignment changes type of `$i` to `T` without taking the *input data-flow* into account, and f_{B_2} is identity function, we can simplify the equations to

$$OUT(B_1) = \{Integer\}$$
$$OUT(B_2) = f_{B_2}(MEET(OUT(B_1), OUT(B_3)))$$
$$OUT(B_3) = \{Double\}$$

$$OUT(B_1) = \{Integer\}$$
$$OUT(B_2) = f_{B_2}(\{Integer\} \cup \{Double\}) = \{Integer, Double\}$$
$$OUT(B_3) = \{Double\}$$

With this result we can, for example, check that function `foo` is invoked with correct argument type, because from $IN(B_2)$ we know that `$i` at the moment of invocation of `foo` can be either of type `Integer` or `Double`.

2.2.4 Finding Solution for the Data-flow Equations

Lattices.

The algorithm for finding a solution to a set of data-flow equations is based on 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. If a and b are elements of a lattice, we denote their least upper bound as $a \wedge b$.

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.

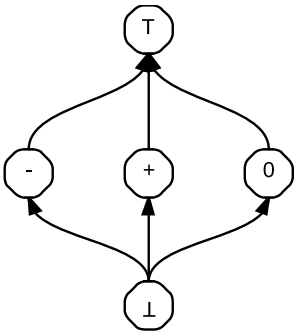


Figure 2.1: Bounded lattice with 5 elements.

There are several properties of lattices important for DFA.

If we have a finite bounded lattice $(S, \leq_s)$ and a function $f : S \to S$ that is monotonous, in order words, $\forall a, b \in S : a \leq_s b \Rightarrow f(a) \leq_s f(b)$, 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.

From this property further follows that if we use a finite lattice's elements as a domain for transfer functions, and the transfer functions are monotonous, then

the set of data-flow equations has a solution, which is a safe approximation of the, in a sense, best solution to the data flow problem. Details can be found in [9].

Product lattice obtained from two or more lattices is also a lattice, which can simplify the design of data flow analyses.

Power set $\mathcal{P}(S)$, the set of all subsets of S, is also a lattice, the lattice order relation being subset relation $\subseteq$.

The Iterative Algorithm

The algorithm to find the solution to the data-flow equations initially sets $OUT(B_i)$ for every basic block to the initial *data-flow*, which should be the lowest element of the lattice. Then it iteratively takes a basic block B_j such that its input *data-flow* $IN(B_j)$ has changed or is not initialized yet and computes $OUT(B_j)$, which might change the input *data-flow* of the ancestors of B_j. The process is repeated until a the system stabilizes.

The algorithm can take basic blocks in any order, however, the *reverse post order* provides the best time complexity [4].

Bit Vectors as Data-flow Representation

The performance of the algorithm also depends on the implementation of *data-flow*, the *MEET* operations and the transfer functions.

Data-flow often represents a subset of a set of possible values and the *MEET* operation is either union or intersection. For example, a subset of all possible types of a variable. Furthermore, we want to calculate the information for all variables not only for one. If we know the number of variables n and types m in advance, we can represent the *data-flow* as a bit-vector where groups of m bits represent a data of single variable and within those bits, value of bit with index i indicates whether the type with index i is in the set. Union or intersection can be implemented using fast bitwise operations.

2.2.5 Abstraction

The *data-flow* value for program point is typically an abstraction of some property of the set of all possible program states that can be observed during real execution for that program point.

Abstracting the desired property is often important in order to make the analysis practical or even feasible. Let us consider an analysis that should determine the sign of integral variables at each program point. We can have inference rules of the following form.

$$\frac{S_1 \vdash v_1 : -10(sign : \ominus) \quad S_1 \vdash v_2 : 3(sign : \oplus)}{[v_3 = v_1 + v_2, S_1] \rightarrow S_2, S_2 \vdash v_3 : -7(sign : \ominus)}$$

However, the implementation would not be very efficient and also the full information about variables values is not always available, 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:

$$\frac{S_1 \vdash v_1 : \ominus \quad S_1 \vdash v_2 : \ominus}{[v_3 = v_1 + v_2, S_1] \to S_2, S_2 \vdash v_3 : \ominus}$$

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

$$\frac{S_1 \vdash v_1 : \ominus \quad S_1 \vdash v_2 : \oplus}{[v_3 = v_1 + v_2, S_1] \to S_2, S_2 \vdash v_3 :?}$$

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:

$$\frac{S_1 \vdash v_1 : \ominus \quad S_1 \vdash v_2 : \oplus}{[v_3 = v_1 + v_2, S_1] \to S_2, S_2 \vdash v_3 : \top}$$

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

$$\frac{S_1 \vdash v_1 : \ominus \quad S_1 \vdash v_2 : \top}{[v_3 = v_1 + v_2, S_1] \to S_2, S_2 \vdash v_3 : \top}$$

It is no surprise that we used symbol $\top$, which is also used to denote the greatest element in a lattice. By adding $\bot$, the least element, and inference rules for it, we can get a lattice depicted in 2.1 and use this abstraction as *data-flow* value.

Formalization

There is a theoretical framework for designing sound and correct abstractions. In a nutshell, an abstraction following this framework has to include:

- concrete domain C, which has to be a lattice ,
- abstract domain A, which has to be a finite lattice,
- concretization function $\gamma : A \mapsto C$,
- abstraction function $\alpha : C \mapsto A$,

In our case the concrete domain could be the power set of all possible integral values, which is a lattice, and the abstract domain with elements $\{0, \ominus, \oplus \top, \bot\}$ was described above. The γ and α functions map values from one domain to another. When we map elements of concrete domain to the abstract domain,

we loose precision due to abstraction. Let us provide few examples, instead of a formal definition of γ and α.

$$\gamma(\oplus) = \{1, 2, 3, ...\} \qquad \alpha(\{3, 4, 5\}) = \oplus$$
$$\gamma(\top) = \top = \{..., -2, 1, 0, 1, 2, ...\} \qquad \alpha(\{3, -4\}) = \top$$

The framework for abstract interpretation further defines necessary conditions for the γ and α functions and conditions for the *transfer functions* with respect to γ and α mapping. Details and other static analysis methods based on abstract interpretation framework can be found in [10], [11].

2.2.6 Intraprocedural Analysis

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

A straight forward approach is to regard other routines as black boxes and assume the worst with respect to how a routine call can change the program state. Nonetheless, in practical programming language with references, or pointers, pass-by-reference arguments, lambda functions with capture by reference, and other caveats, an innocent routine call can theoretically do almost anything to the program state even from the local point of view of the function we are analysing.

Context Sensitive Intraprocedural Analysis

The most precise solution is to analyse a function, say `foo`, for each possible calling context. In other words, re-analyse `foo`'s body every time we encounter its invocation when analysing another function, and use the program state before the invocation of `foo` as an initial state for the re-analysis of `foo`.

Caching of the results of analysis based on the calling context, so that we do not re-analyse it if the context is the same as some context previously encountered, is possible. However, in the most generic form, it is impossible to asses the equality of program states, because the context consists not only of arguments passed to the function, but also the global state including the heap.

Another problem, which makes fully context sensitive approach impractical or even impossible, is polymorphism or dynamic invocation of routines in general, be it lambda functions, virtual methods, reflection, or any other means. It is not known statically what function will be invoked and thus what function to analyse. The call site can be determined by the analysis itself, but in general it cannot be guaranteed.

Heap Abstractions

When a routine takes a pointer or reference to some object on the heap as an argument, it can traverse to other objects on the heap and change their state. Let alone low-level languages that allow direct manipulation with the memory and can therefore change anything in the heap.

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

The heap can be important part of the program state for some kinds of analyses. If we want to take the heap state into the consideration during the analysis, we need an abstraction for the heap to make the approach feasible in practice.

Researchers have proposed several approaches to representation of heap data in an abstract way to minimize memory requirements and provide fast algorithms to check two heap abstractions for equality or isomorphism. A sound and useful definition of heaps isomorphism is also a problematic task by itself. The paper [12] provides a survey on the heap abstraction models.

Region Based Analysis

Another approach is to analyse a routine once in a generic context setting and create a *transfer function* that summarizes the effects of the routine call to the program state [4]. However, in practical setting, we need to decide how to derive the *transfer function* and how to represent it. A constant *transfer function* that always returns the $\top$, in other words, the worst assumption about the program state, is always safe *transfer function* for any routine call, but we have not gained anything over the naive solution from the beginning of this section.

In the case of Control Flow for Phalanger, we used a variant of a region based analysis. Details are discussed in the following section.

2.3 Control Flow for Phalanger Approach

2.3.1 Assumptions

The approach taken in Control Flow for Phalanger is based on an observation that even when developers are given a dynamically typed programming language, it does not mean that they will write dynamically typed programs. Empirical evidence for this observation was presented in [13].

We are therefore assuming that large part of the analysed code corresponds to a statically typed code one-to-one, it only does not include the type information. However, we do not want to ignore the dynamic typing completely.

Another decision we made, is that only local variables will be a viable target for compiler optimizations use case, since the compiler optimizations have to be safe with respect to all the possible and obscure corner cases, and global variables and heap memory are difficult to analyse precisely. On the other hand, with bug-hunting and integrated development environment supporting style analyses, we can afford more courageous assumptions.

In this section we focus on the type analysis, because it is by far the most complex kind of analysis implemented. Other analyses, like constant propagation, follow the same concepts and, furthermore, they rely on the results of the type analysis and points-to analysis, which is also discussed in this section.

2.3.2 Analysis Results

The outcomes of the analysis of a single routine are:

Variables table: type information for local variables – for every local variable, a set of its possible types at any point during the execution of the rou-

tine. Note that it summarizes all types that variable can have at different program points. Knowing that a variable has only one possible type can be useful for compiler.

Type information for expressions – every expression in the routine's body will be annotated with the type or set of types the expression can evaluate to.

Warnings – some expressions expect only operands of certain type; if the analysis encounters such expression and from the analysis it follows that the operand is not of the expected type, the piece of code in question is reported as a warning. The same holds for the type expectations extracted from documentation comments for fields, routines and others.

Another outcome of analysis across all routines will be *globals table* with type information for global symbols that is global variables, instance and static fields and routines.

2.3.3 Local Analysis: Overview

We can think about expressions in terms of evaluation trees, like in figure 2.2. Since expressions should be annotated with their type information, the analysis can use a bottom up approach and annotate the leaves of the evaluation tree first and then recursively from the type information of operands infer type information for compound expressions.

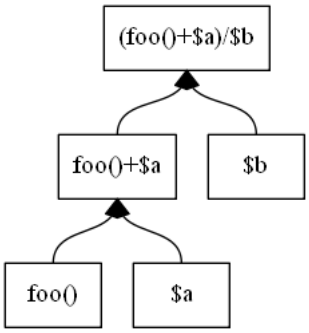


Figure 2.2: Evaluation tree of an expression `(foo()+$a)/$b`.

Statements or expressions that can change the program state, like assignment statement, will have their operands annotated with the type information and thus have all the necessary information to reflect the change of the program state accordingly. For instance, the type of the right hand side expression of an assignment statement will determine the new type for the variable on the left hand side.

Some type information can be inferred from the code without any knowledge about variables' types. For example, the result of string concatenation is always a `string`, no matter what the types of the arguments are. However, simple local variable use is an expression and we need to know the possible types of the variable to annotate this expression with its type. For this, we need to perform a DFA that will give us the type information for each variable at each program point. Note that the *data-flow* values are used only temporarily to annotate the

expressions and build the variables table, but we do not create actual *data-flow* instance for every program point.

The questions that have to be answered, in order to implement the DFA, include:

- what the *data-flow* values will be,
- what data structures will be used to represent them efficiently,
- how to deal with routines calls and their effects to the program state.

2.3.4 Local Analysis: Data-flow Values

The *data-flow* should capture the type information for variables within the routine. We do want to support dynamic typing, therefore for every variable, we have a subset of all its possible types, not only a single type. The *data-flow* value is a map from variable names to subsets of types.

For an efficient representation of the *data-flow*, the routine's body is scanned for all local variables names referenced types prior to the analysis. After the scan, the number of variables and types is a known constant and the *data-flow*, being an array of subsets of a set of all the types, can be represented as a bit-vector. The MEET operation is a union, because if a variable can have type `T` from one branch and type `K` from another, at the join point we can only assume that it has either type `T` or `K`.

However, with a real world class based object oriented programming language, the situation is not as simple. We have to deal with subtyping, and we also want to support type information from documentation comments, which nonetheless should be distinguished from the type information inferred from the actual code.

The basic *data-flow* values for a single variable and set of types $\{int, false, null\}$ are depicted in the figure 2.3. Aside the whole set of all types references within the routine's body, we have another artificial value called *Any*, which simply designates any possible type, not restricted to the set of known types. In the following paragraphs, we discuss how we added support for classes and documentation comments to this schema. We focus on *data-flow* lattice for a single variable; the lattice for all variables will simply be a product lattice.

Booleans

In the case of booleans, we distinguish `true` and `false`, because it is a well known pattern in PHP for routines to use `false` return value as an indication of failure. Routines then can have, for example, have type `false|object` meaning that this routine returns either object or value `false`, never value `true`.

Null

Another thing to note is that we treat `null` as a special type. This again comes from PHPDoc documentation comments, where one can, for instance, state `integer|null` as return type of a routine, in which case representing the routines return type as just `integer` would not be precise. Moreover, having `null` can be used to distinguish between non-nullable and nullable reference types. Control Flow for Phalanger by default assumes that all the reference types can be `null`,

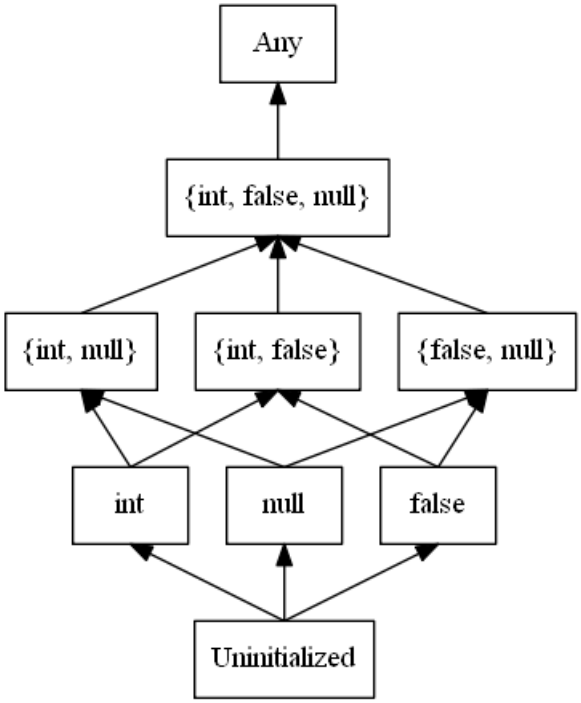


Figure 2.3: Basic lattice with types.

but it can be configured to assume the opposite, in which case it can, for example, report type error if a routine expects instance of some `object` as an argument, but is given `null`.

Type Hints

We refer to the type information gathered from documentation comments as type hints. For example, routines can have documentation comments that state expected types of the parameters. However, it is perfectly legal to invoke the routine with actual parameters of different types, so we cannot rely on the documentation completely, but we want to utilize it.

For this purpose, we distinguish between type information that was inferred from the code and should always be valid and type information that was extracted from the documentation comments or possibly other unreliable sources, for instance, return type of a non-final method, which cannot be accurately determined because of polymorphism.

The distinction is realized as a single bit flag we call "type hint". It is a single flag for the whole set of possible types, so we cannot have a set of two possible types where one is "type hint" and the other is not.

We can think of all the sets with "type hint" flag as a parallel lattice to the lattice of sets without "type hint" flag. Let us say that $hint(x)$ denotes an element from the parallel lattice corresponding to element x in the original one. If we are comparing two elements a and b', where $b' = hint(b)$ is from the "type hint" lattice, we say that their upper bound or supremum is equal to $hint(a \wedge b)$, formally $a \wedge b' = hint(a \wedge b)$. A lattice that is formed from putting together the original types lattice and the "type hint" lattice for types $\{int, string\}$ is depicted in figure 2.4.

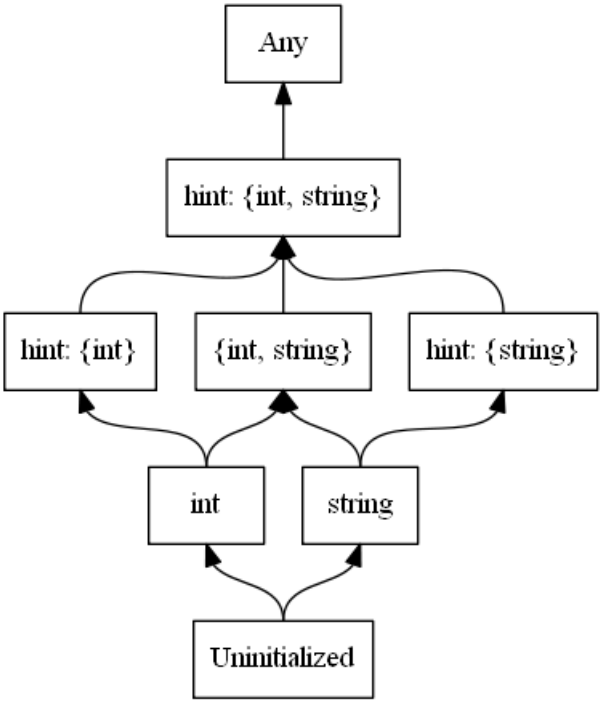


Figure 2.4: Types lattice with "type hint" flag.

Classes

We refer to built-in types as basic types, those are

- int
- double
- string
- resource
- false and true (instead of boolean)
- null
- array
- callable – for example: lambda

If we say that variable `$a` is of type `T`, where `T` is a non-final class, even in statically typed language, this could mean that `$a` can be in fact instance of many different classes, namely `T` and all its subclasses.

Because we are in a dynamic environment, we cannot, in general, know all subclasses of `T` in advance. Nonetheless, it can be useful to distinguish between "`$a` is of type `T` only" and "`$a` is of type `T` or any of its subtypes".

To make this distinction, we add another flag "subclasses", which works similarly to "any type" flag: it holds or does not hold for the whole set of class types. Also the upper bound of two elements, where one of the has the flag "subclasses", will has the "subclasses" flag.

Moreover, we add one artificial value, which is `object`, and it denotes arbitrary class instance, but it is not the same as *Any*, because it excludes the other primitive types. A lattice for just class types $\{T, K\}$ is in figure 2.5.

The upper bound of any set of class types and a set of basic types is a union of those. However, the upper bound of a set with some class types in it and a set with `object` in it is a union of the basic types and `object`, so the class types are "overridden" by the `object`. This is illustrated in figure 2.6.

This lattice with class types can be amended with the "type hint" flag the same way we did before we introduced classes.

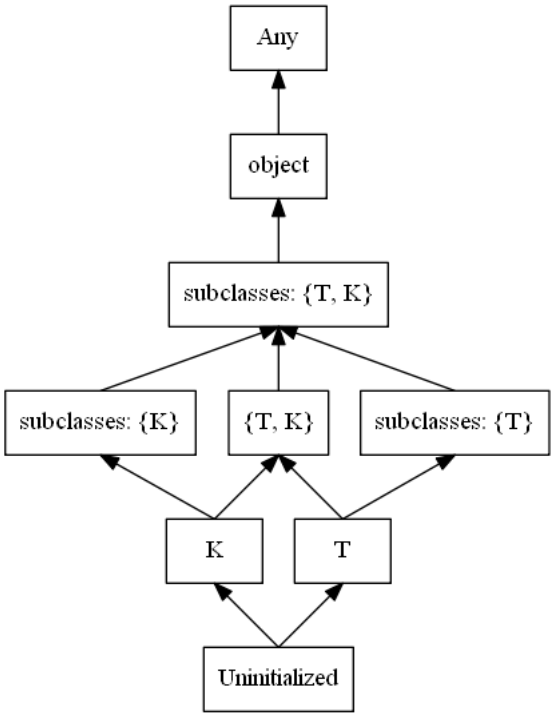


Figure 2.5: Types lattice with classes.

2.3.5 Intraprocedural Analysis

The approach we described so far considers only local variables, but we want to analyse object and class fields' types, and global variables. We also have to take into account possible changes to local variables made from other routines through references.

The assumption here is that the results of analysis for fields and global variables do not have to be precise and that although PHP is a dynamic language that permits dynamic fields and dynamic methods, mostly the developers do not abuse those features and we can assume that instances of the same class have the same fields of the same type and the same methods across the whole program. Furthermore, routines return type and expected arguments types will mostly be constant and independent from the program state or each other.

Routines Return Values

When the analysis encounters a routine invocation, it needs to know the routine's return type and possible other pieces of information we discuss later.

If the source code of the routine in question is available, in other words, it is part of the analysed source, we recursively trigger its analysis in order to infer its return type. When the analysis finished, we save the results to the *globas table* so that we do not need to analyse the routine next time and resume the analysis of the original routine.

This approach, however, does not work for recursive and mutually recursive routines. The solution we use is that if a cycle in routines invocations is detected, we do not analyse the last routine in the cycle, but instead we assume the worst about its return type – we assume it can return any type.

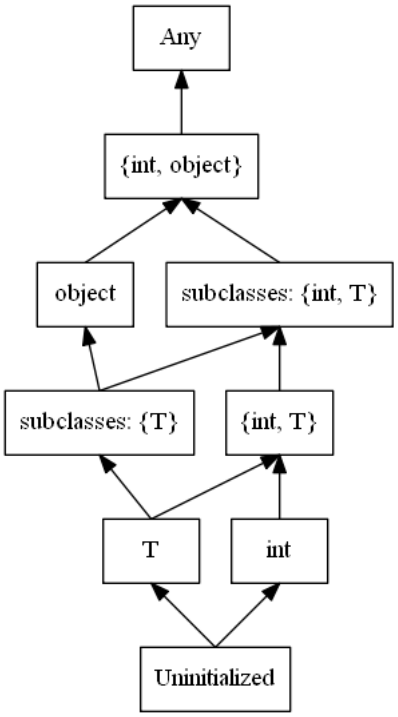


Figure 2.6: Types lattice: class and a basic type.

Heap

Let us first provide an example, where heap abstraction and context sensitive intraprocedural analysis would be useful:

```
function foo() {
  $a = new X();
  boo($a);
  return $a->bar;
}

function boo(X $a) {
  $a->bar = 3;
}
```

Here the function `boo` changes type of the nested field `bar`. If we do not reflect this in the program state after the invocation of `boo($a)` in function `foo`, we cannot determine the type of the return expression `$a->bar`.

However, we decided to approach this problem differently. We do not analyse individual object instances, instead we summarize the type information inferred for individual instances of the same class and provide those results within the *globals table*.

Illustrated on the same example, when analysing function `foo`, the analysis would encounter the invocation of `boo`. Since `boo` has not been analysed yet, this triggers analysis of `boo`. While analysing `boo` the *globals table* learns that field `bar` can be of type `integer`. When analysis of `boo` finishes and analysis of `foo` resumes, on the line with `return $a->bar` we know from the *data-flow* that `$a`

is of type `X` and from the *globals table* we get that field `bar` of class `X` can be of type `integer`.

With this approach we do not have to model heap, which can be more effective, but on the other hand, there are cases different to this example, where our approach fails to devise any useful information and heap model, could give us better results. For instance, if function `boo` did not provide type information for its parameter in its signature, because we analyse each routine in a generic context.

Moreover, we ignore the effects of indirect assignments and references to the *globals table*, which means that the *globals table* is not safe approximation. However, as we stated at the beginning of this section, the analysis of instance fields is meant for bug hunting purposes and we can afford some imprecision. In addition, the type information inferred by means of *globals table* is marked with the "type hint" flag.

The global variables and static class field are analysed in the way as instance fields.

Type Documentation

There are two different cases where we need to consider type information for fields or global variables: when a field or global variable is read, and when a field or global variable is assigned to.

Field and global variables can have PHPDoc type documentation. In the case of assignment, we want to check that the value assigned to the variable has a type compatible with the documentation, otherwise it is possibly a bug or documentation error.

When it comes to type annotation of an expression that uses a global variable or a field, if the type documentation is available, we use it over the information from *globals table*, but still mark the type information with "type hint" flag.

References Analysis

References in PHP are used, but occasionally, therefore we do not want to ignore them, but we do not require a precise results when references are used in the analysed routine.

Another piece of information the analysis needs to know about a routine that is being invoked from within the analysed routine is which arguments are passed by reference to it, so that their value can be changed inside the invoked function. This is fortunately a part of the routine's signature, and can be easily determined. We simply throw away any assumptions we had about the type of a variable which is being passed by reference to another routine.

The situation is more complicated with references within a single routine. When a statement or expression can change the type of a variable, it can change type of any variable that this variable points to, if it is a reference.

We use a simple approach: prior to the analysis, the routine body is scanned for any assignments by reference (`$a=&$b`) and two sets are created: variables that can be a reference and variables that can be pointed to by a reference. This is very conservative approximation, but we expect sparse usage of references.

When the analysis finds out that a type of a variable can be changed to `T`, it checks if the variable can be a reference and if so the types of all the variables that can be pointed to by any reference are updated. Those variables **may** only be pointed to, but do not **have to** be pointed to, so we take their current type information and merge with type `T`, creating over approximation.

Moreover, any routine's invocation can change any local variable from the can be pointed to set, because references can "leak" on the heap, for where the other routine can read them. That is why, after any routine invocation, we throw away any assumptions about variables in the can be pointed to set.

The same holds for local variables that are captured by reference by some lambda function. The lambda function can again "leak" on the heap, from where the other routine can invoke it.

3. Related Work

A brief overview of the static analysis methods was presented in the previous chapter. In this chapter, we focus on tools that also use static analysis to analyse PHP code for different purposes. We also shortly mention interesting tools for other dynamic languages.

3.1 Security Vulnerabilities in Web Applications

Most of the existing work on static analysis of PHP is focused on discovery of security vulnerabilities in web applications that typically come from improperly handled user input, also called taint-style vulnerabilities. It is important for such analyses to be able to follow the flow of data from global variables like `$_POST` that contain user input, therefore more precise model of heap memory is required so that flow of data in between object instances and routine calls can be analysed.

An analysis for security vulnerabilities has also a different model of usage. Such analysis can be run less frequently, for example, only before release or as a part of a continuous build process. Interactive on-the-fly analysis in an integrated development environment could also be a viable use case, but typically not the main goal. Moreover, such analysis is not likely to be run every time the application is to be compiled or interpreted.

Some of the available tools for detecting taint-style vulnerabilities in PHP are Pixy[14] and recently released Weverca: Web Verification Tool[15].

3.1.1 Weverca: Web Verification Tool

Weverca is an implementation of static analysis framework also based on the Phalanger parser. As opposed to Control Flow for Phalanger, the main goal of Weverca is to provide security vulnerabilities analysis, although it is capable of supporting other kinds of static analyses.

Memory Abstraction

Weverca represents the program state at each program point by an abstraction of the complete memory state including local variables, global variables and static fields. Compared to our approach, we represent the program state by the state of local variables only and global variables and fields not analysed precisely in context sensitive way, but summarized in one global database shared among all analysed functions.

The approach of Weverca enables better precision and their default implementations of the memory model do provide such precision. On the contrary, our approach permits more effective representation of the program points state. Needless to say, both tools provide means to be extended with an implementation of the other approach.

Moreover, the memory abstraction used in Weverca includes defined symbols, such as routines, classes and others. PHP permits to define symbols dynami-

cally and in certain circumstances symbol cannot be used before it is defined[1]. Therefore Weverca is capable of discovering use before declaration kind of errors for global symbols. In Control Flow for Phalanger, we decided to not support this, because most of the modern object oriented PHP projects use *autoloading* and with autoloading it is impossible in general to analyse which files are being imported at which program points. Autoloading rules are often simple and follow similar patterns and so a viable possibility for a static analyser would be to let the user choose from predefined set of autoloading rules that the analyser understands. However, neither of the tools implement this feature yet.

Intraprocedural Analysis

In order to make the interprocedural analysis context sensitive, Weverca inlines the control flow graph of invoked routine in the control flow graph of the analysed routine. Note that there can be more than one routine that can be invoked due to polymorphism or dynamic nature of PHP. In such case Weverca inlines all of them adding a non-deterministic choice between them, in other words, edges from the routine call program point to the first program point of all the possible routines.

As discussed in the previous chapter, Control Flow for Phalanger uses modular approach, which may scale better, but lacks the precision of Weverca.

Type Information

Support for type hints. Enough for whole subsubsection?

Design and Implementation

From the point of view of implementation, Weverca uses Phalanger as a parser, but the design does not evince any intention of tighter integration with Phalanger. The version of Phalanger used is slightly outdated, and thereof support for newer PHP language constructs is probably missing.

PHPDoc support.

Intermediate Representation

3.2 Type Inference

Type inference for dynamic languages is typically implemented for the purposes of compilation or interpretation. A notable implementation is type inference for PHP in Facebook's Hip Hop project [16], which is a compiler from PHP to C++ and a custom intermediate language that can be run in the Hip Hop virtual machine. Hip Hop performs type analysis in order to find a single type for a variable, so it can treat it as statically typed variable during compilation. However, if a single type for a variable cannot be determined, Hip Hop does not analyse the type information any further and fall backs to the dynamic typing.

[1] A symbol cannot be used before the file with its declaration is imported, but symbols from the same file can be used before they are declared.

There are implementations of type inference for other dynamic languages. Ecstatic[17] is type inference for Ruby implemented using control flow insensitive cartesian product algorithm. Rubydust[18] introduces a *constraint based dynamic type inference* that infers static types based on dynamic program executions.

3.2.1 Phantm

Phantm[19] is a tool for detection of type related errors. From all the projects mentioned in this chapter, the aim of Phantm is closest to our project, which is why we also used Phantm for evaluation and compared its results with ours in section 5.1 Comparison to Phantm.

Phantm uses semi-dynamic and semi-static analysis approach. The web application in question is run up to a defined point, which is invocation of special Phantm's function that collects data about the state of the application, especially, values of global variables. This data is then used as an initial state for static analysis. The dynamic part of the analysis is called bootstrapping. This design illustrates that although type related errors can be searched for in generic frameworks, libraries or, for example, command line utilities written in PHP, Phantm's focus is on complete web applications.

- *similar type abstraction (null, false, true, int values are repr. explicitly for const. prop.)*
- *heap abstraction, but assumption about routines only changing a local region in heap and returning fresh instances.*
- *array support: merging, termination – limited nesting.*
- *ignores completely:*
  - *references*
  - *indirect accessed (variables, fields)*
  - *assignment in conditional exprs.*
  - *eval, autoload*

4. Design and 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) [4] structure. This structure is then traversed by the back-end using the visitor design pattern [20]. 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 minor 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 could not 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. The 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, for example, 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.

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 results of an 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 results of an analysis, for instance, results of the 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.

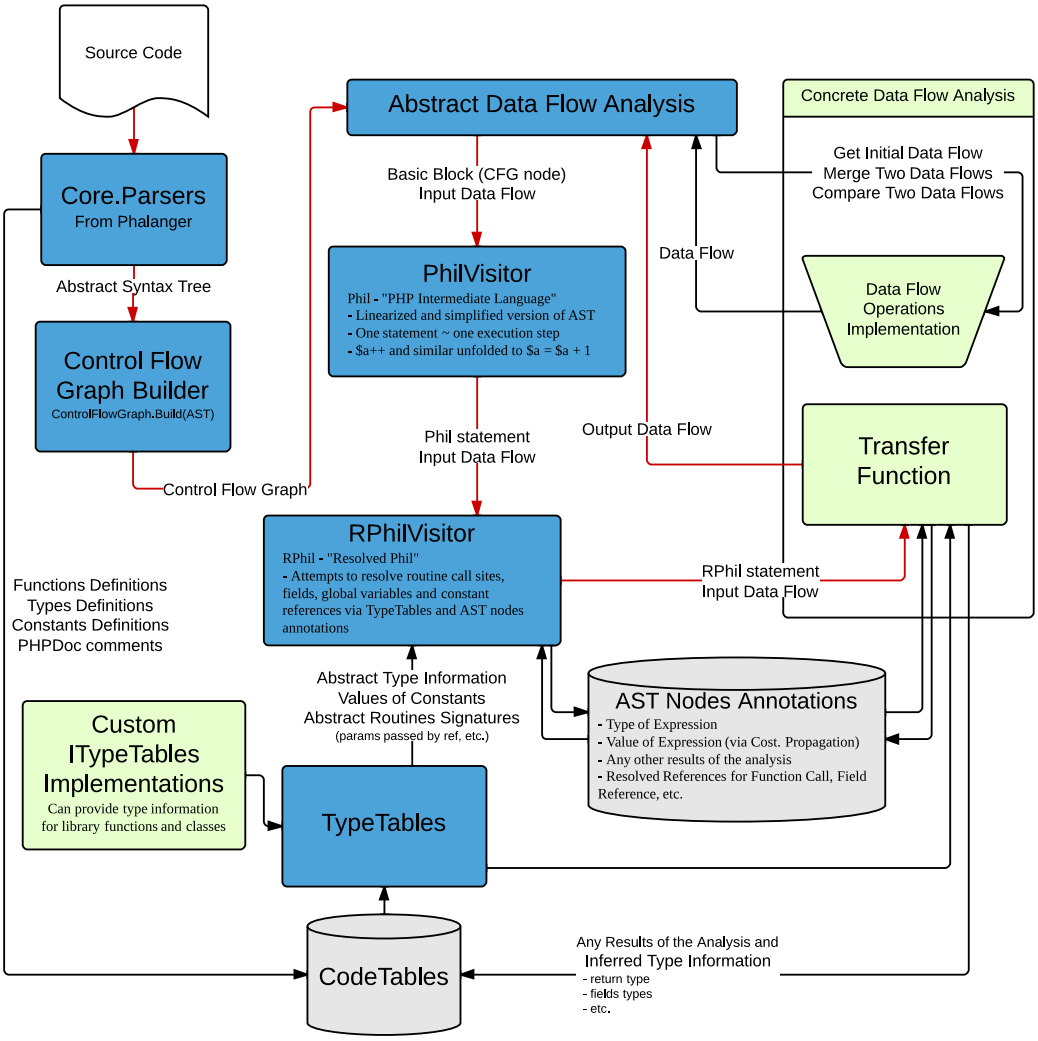


Figure 4.1: Control Flow for Phalanger Design Overview

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 they are passed 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, and so on. 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.

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 Languages and Control Flow Graph

The aim of the intermediate languages is to simplify the interface for concrete analyses implementations. However, at the same time, it was desirable to stay as close to the original AST elements as possible in order to easily propagate the results and to easily integrate Control Flow for Phalanger into the Phalanger project. Lastly the intermediate language should stay close enough to PHP code so that PHP specific patterns can be recognized.

4.3.1 Phil: PHP Intermediate Language

Motivation

The main aim of Phil is to provide a framework for traversing the AST elements in the order of their execution in the smallest execution steps possible with respect to their possible effects to the environment. As opposed to implementing a full blown AST visitor for every analysis, Phil helps to avoid repetitive code that implements the AST structure traversal and breaking down some of the syntactic constructs like `$i++`. As discussed later, a "Phil element" is often just a conceptual term, but in reality most of the Phil elements are represented directly by AST elements in order to save resources and stay close to the original AST.

Implementation

Phil is implemented mainly by two classes `PhilVisitor` and `Linearizer`, which is a class nested in `PhilVisitor`. The `PhilVisitor` class implements a variant of the visitor design pattern with virtual `Visit*` methods for all the Phil elements. The `Linearizer` is implementation of a visitor for the AST elements and it converts them to Phil elements.

An important difference between original AST elements and Phil elements is that Phil does not have a hierarchical structure. Therefore an implementation of the `PhilVisitor` class does not need to perform recursive traversal for every visited element.

Under closer look, most of the AST elements correspond to exactly one Phil element[1]. Those AST elements are passed to `PhilVisitor` as they are. So in this case the `Linearizer`, as the name suggests, performs only the traversal and linerizes the AST structure for `PhilVisitor` so that it does not have to care about AST structure traversing.

Elements that have to be unfolded into several operations, such as `IncDecEx` representing post or pre increment or decrement, have to be represented by more elements. One option would be to create a whole new alternative AST structure that would correspond the unfolded code. In our approach, we reuse the AST element to represent one of the unfolded operations and create new special Phil elements that wrap the original AST element and their only purpose is to indicate the stage of the compound operation. So for example, `IncDecEx` is broken down to

- `Expression` that represents the variable access and is recursively broken down to Phil elements,
- `IncDecEx` that represents the binary operation, so the `VisitBinaryEx` method is invoked for it on `PhilVisitor` as for any other binary operation.
- `IncDecPhilAssignment` that represents the assignment, and the `VisitAssignment` method is invoked for it on `PhilVisitor` as for any other assignment.

Because we have an object instance to represent every Phil element, we do not have to visit them on the fly, they can also be saved into and array and visited by the `PhilVisitor` later. This enables us to perform the traversing only once, shall

[1] or they are ignored, which is the case of most control flow changing statements and declaration statements

it be a performance issue in the future. It is also possible to crate accurate control flow graphs when it comes to try-catch blocks, because a single AST element, can cause exception in any stage of its evaluation, therefore it should be split into single execution steps and those put into separate basic blocks so that an edge from each of them to the catch block can be created. Note that the design is prepared for this case, but it is not implemented yet.

Variables Accesses and Context Information

The AST is broken down to small pieces visited by the `PhilVisitor`, which does not have to take care about their structure or order. However, sometimes a context information is required in order to handle some elements properly. This is the case of expressions that can be used as left hand side of an assignment or as a parameter passed by reference. In those cases the `Visit*` method of `PhilVisitor` takes another parameter of type `VarUseContext` that identifies the context in which the expression is accessed.

To simplify the AST further, all the accesses to a memory location, be it local variable access, instance field access, or any other, are wrapped in an implementation of abstract class `VarLikeConstructInfo`. This class provides unified interface for typical operations performed on this kind of elements. Namely, for example, `GetMayReferenceVars`, which returns local variables that this access can point to (typically using the results of points-to analysis).

4.3.2 RPhil: Resolved PHP Intermediate Language

RPhil elements are the same as Phil elements, but with additional information retrieved from Type Tables where possible. `RPhilVisitor` implements a variant of the visitor pattern in the same way as `PhilVisitor`, but the virtual `Visit*` methods take additional parameters that provide this kind of information.

A simple example is direct static field access, in which case the resolving is trivial, because the `RPhilVisitor` can get the class and field's name directly from the AST element. However, even a direct object's method access, is challenging, because we do not know the type of the expression that is being dereferenced using `->` operator.

RPhil does not perform any analysis in order to resolve such cases, but by convention it looks for certain additional attributes attached to the AST elements in question. Those attributes are expected to be filled in by analyses. Concretely, every expression can be annotated with its type and value.

4.3.3 Control Flow Graph

Control Flow Graphs are constructed by an implementation of the AST visitor. The class holds an instance of a basic block that is being currently constructed. Statements are sequentially visited, and those that do not change the flow of the control are just added to the current basic block. Statements that change the flow lead to a creation of new basic blocks that have incoming edges from the previous basic block.

This structure works for well structured code, but PHP permits `goto` statements, especially forward ones are problematic, because at the moment the `goto`

statement is being processed, the basic block for the target might not be constructed yet. For this purpose a table of `goto` statements and all the labels is created during the control flow graph construction and the `goto` statements are connected to their `labels` at the end once the whole routine body is processed.

Exceptions

When a try code block is processed, every statement is placed in its own basic block that is connected to the basic block of the consecutive statement and connected to all the possible catch code blocks.

The possible catch blocks are chosen pessimistically, so a statement in try code block is connected to all the catch code blocks, even nested ones, up to a catch block that catches generic `Exception` or to the graph's *exit* basic block.

Edges

Control Flow Graph edges can have an optional attribute which states the expression that has to hold if this edge is taken during the execution. For example, an edge to then branch of `if ($x==3)` will have expression `$x==3` and the analyses may work out from it that `$x` is equal to `3` in the basic block corresponding to the then branch.

4.4 Data Flow Analysis

A data flow analysis (DFA) in general can be performed on any graph, the implementation of Control and so even in our case, we did not want it to be tied Flow Graph. For this purpose, our implementation of DFA is performed on interfaces that Control Flow Graph implements, but they can be implemented, for example, by definition-use graph[4] and DFA can be run on this graph too. The interfaces are depicted in figure 4.2.

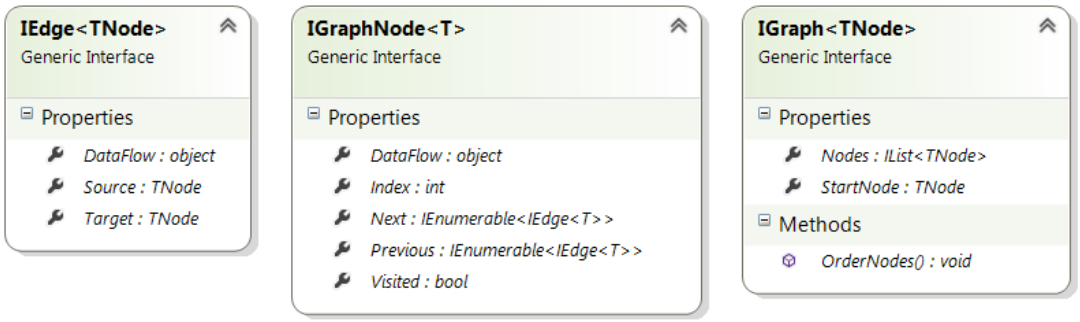


Figure 4.2: Generic Graph Interfaces for DFA

The generic data flow framework handles the order in which the graph nodes should be visited, compares the input and output data flows and decides when the analysis has reached a fix-point. However the concrete type of data flow, operations with data flow instances and the transfer function are left to be defined by a concrete analysis. In Control Flow for Phalanger, two types of analysis can implemented. The basic one processes only nodes; the "branching" one also

processes edges, in which case it can take the branching expressions of Control Flow Graph into account, for example, but in general any information that the concrete implementation of `IEdge` can provide.

The operations with data flow objects could be carried out by the objects themselves, but this would mean that already existing classes that happens to be suitable for being a data flow would have to be wrapped. And also one data flow representation, could not have different operations for different analyses. An example of this is `BitVector` class from .NET class library: it cannot implement any additional interface, and some analysis perform union of two vectors as the meet operation, while others perform intersection.

Nonetheless, having the data flow objects implement the operations by themselves is more convenient and allows better encapsulation. Because Control Flow for Phalanger is meant as a framework for as well as software on its own, both scenarios are supported and some convenient generics based implementations of required interfaces are provided. The whole design is captured in diagram 4.3.

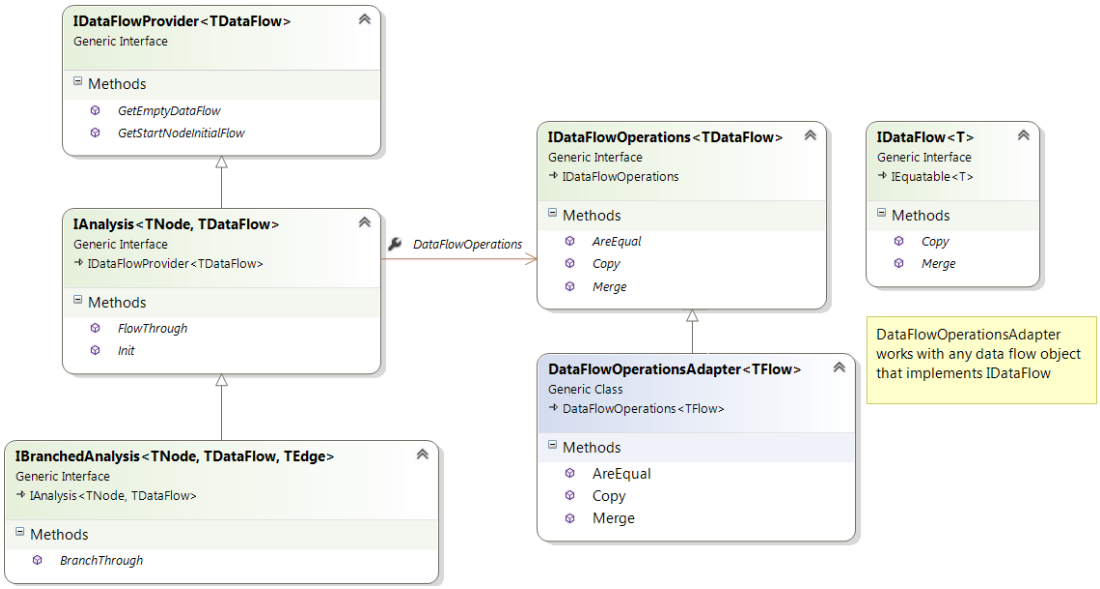


Figure 4.3: Interfaces for concrete Data Flow Analyses

A simple example of concrete Data Flow Analysis is the built-in constant propagation analysis, which is discussed in one of the following sections.

4.5 Tables

Tables module responsibility is to maintain a database of symbols that can be referenced from the code. This module distinguishes between two kinds of symbols: symbols that can be found in the analysed code, and external symbols, which could be library functions, for example.

4.5.1 Type Tables

The most generic interface `ITypeTables` for querying the tables database merges the two kinds of symbols and provides mainly abstract type related information.

The structure is shown in class diagram 4.4.

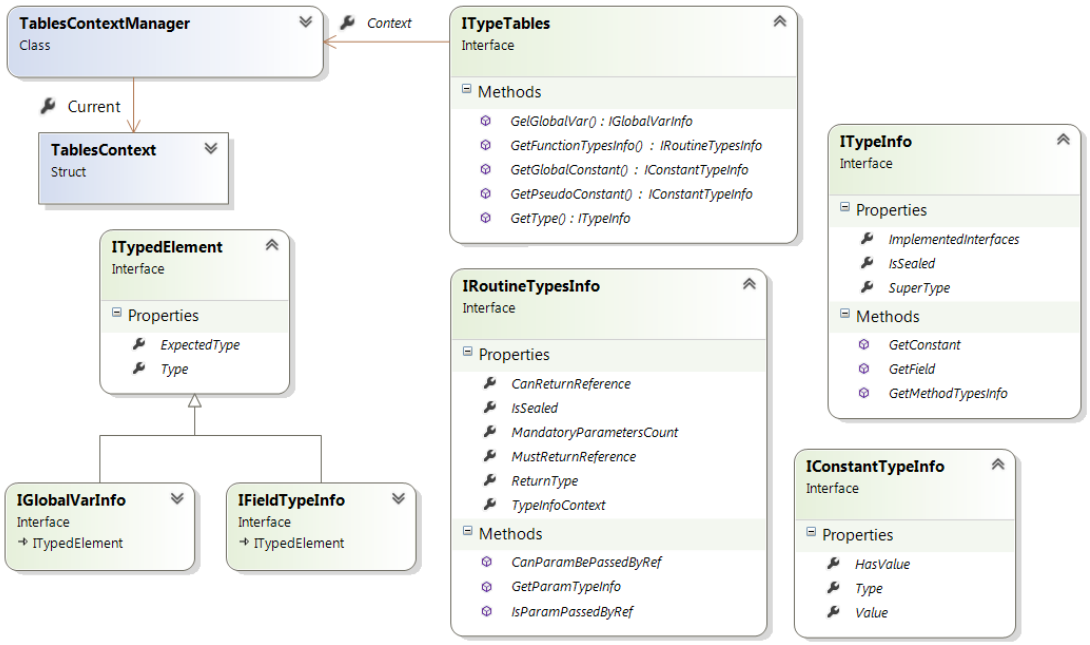


Figure 4.4: The Interface of Type Tables

The tables need to know a context from which the query is made, because certain names can refer to different symbols in different namespaces for example. The `ContextManager` class maintains the current context and the consumers of the `ITypeTables` should invoke `EnterContext` and `LeaveContext` on its `ContextManager` to provide the correct context for their queries.

Global variables and fields provide two pieces type information: `ExpectedType` is checked when a value is assigned to that variable and the `Type` is used as the type of an expression that accesses the variable. Typically, the expected type is what the PHPDoc states and the "type" can be either what PHPDoc states or type inferred from all the assignments made to that variable.

The fact that results of `ITypeTables` queries are only abstract type information not related to a concrete element in code, allows us to support built-in functions and classes and also to handle conditional declarations by merging the type information from all the occurrences found. However, this has not been implemented, but 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.

The basic implementation of `ITypeTables` is `Tables` class, which follows a variant of the composite design pattern. It gets a list of other `ITypeTables` implementations as its constructor parameter and queries those one by one until the symbol is found, or returns a value that indicates that the symbol was not found. The actual implementations of `ITypeTables` include Code Tables discussed later and can include user defined `ITypeTables` implementation that provides information about built-in functions and classes.

4.5.2 Code Tables

Code Tables provide information related to concrete elements in the analysed code. So in this case not abstract type related information is provided, but a reference to the AST element with the declaration. Another difference to Type Tables is that Code Tables can return more results for one symbol name, which is because of conditional declarations. Other than that, the interface is similar to the one of Type Tables.

4.5.3 Dependency Resolver

Dependency Resolver serves as an adapter of Code Tables interface to the Type Tables interface. However, in the case of routines, the routine's declaration AST element does not provide all the necessary type information straight away. Especially inferred return type, which is results of Type Analysis, will not be available if the analysis has not been performed yet.

By convention, all the necessary information for analysing a routine and the results of the type analysis are stored in an instance of `RoutineContext` class in the additional attributes of the routine's declaration AST element. Since this AST element is returned by Code Tables, Dependency Resolver can check if the element has the annotation and if so return it as the result, and if not, it can start analysis of the routine. However, there can be cyclic references between routines, so Dependency Resolver also maintains a list of all the routines that are currently being analysed and if a routine is already in the list, default type information is returned as the query result instead of performing the analysis.

4.5.4 RoutineContext

`RoutineContext` class provides information about a routine needed for most of the analysis in order to analyse the routine. All the local variables, parameters and referenced global and static variables are numbered and accessed through their number instead of their name. The data `RoutineContext` provides are:

- Lists of
  - all referenced types in the routine's body,
  - all referenced variables (local, global, static).
- Results of the points-to analysis: lists of all variables that
  - given variable can point to,
  - can be pointed to by another variable,
  - are captured by reference by some lambda function.
- Control Flow Graph.
- Signature information – implementation of `IRoutineTypesInfo` from Type Tables.
- Results of Type Analysis:
  - inferred return type.

4.6 Analyses

4.6.1 Dead Code Elimination

The Dead Code Elimination is based on the Reverse Post Order algorithm, which is supposed to order nodes in a way that is the most beneficial for Data Flow Analysis and has to be done anyway in order to perform Data Flow Analysis.

Note that the Reverse Post Order algorithm as a part of the Data Flow Analysis module is implemented in a generic way for any `IGraph` implementations.

The algorithm performs a graph search from the *Start* node and after it finishes, the unvisited nodes are at the end of the list with all the nodes. At this point, the list with all the nodes is traversed from the last element and the unvisited nodes are removed until a first visited node is reached.

The Control Flow Graph is design allows to mark some edges as "not executable" if the branching condition is always false. Such edges will be ignored when the Dead Code Elimination is performed, however, the tagging of edges with false branching condition has not been implemented in the final version.

4.6.2 Constant Propagation

Constant Propagation represents a simple example of non-branched implementation of a concrete Data Flow Analysis.

The lattice for the data flow values of single variable is depicted in figure 4.5. The Data Flow type is class `ConstantPropagationDataFlow`, which wraps an array that contains the value of each variable. Note that the variables names are already converted to numbers by `RoutineContext` and those numbers serve as indexes into this array. The value is an `object` instance, so it can be `null`, which is the least element of the lattice, or its value can be concrete singleton object instance that by convention represents `NotAConstant`, which is the greatest element of the lattice.

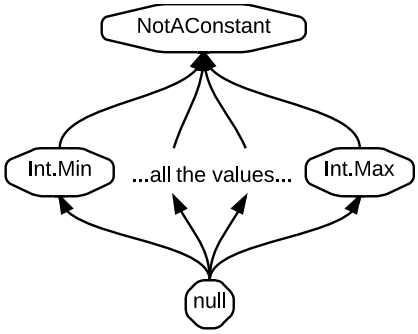


Figure 4.5: The Lattice for Constant Propagation

The transfer function annotates every expression with its constant value if possible. An access to a variable is annotated with this variable's value from the data flow, some expressions can be symbolically executed using the annotations to get operands values, and constants are annotated with their value retrieved from Type Tables.

The data flow is updated only for assignment statement, reference assignment statement and for function call, because some local variables can be passed by

reference, which can change their value. The right hand side of the assignment is an expression, which should already by annotated with its value that is used for the data flow value.

4.7 Type Analysis

The theoretical side of workings of the type analysis was discussed in section Control Flow for Phalanger Approach. Here we focus on practical side of the implementation: the *data-flow* representation and how we realized the annotation of expressions with type information.

4.7.1 Type Information Representation

In this paragraph we discuss representation of type information for a single variable or a single expression. The representation is based on the type information lattice described earlier. It is a set of possible basic types, class types and `object`. It can have flags "subclasses" and "type hint". There are few things to note:

- `object` represents any object type,
- *Any* is a special *data-flow* value that represents any possible type,
- empty set $\emptyset$ is a special *data-flow* value that represents uninitialized type.

The aim is to represent the type information efficiently, because we want to annotate every expression with type information and because we want a memory efficient representation of the *data-flow*, which is an array of type information for all variables.

The type information is always tied to a concrete routine: it is either annotation of one of the expressions in the routine's body, or type of one of the local variables. For every routine we create a special "context" object with a list of all referenced types $\mathbb{T}$. This way every type has a unique index. Let us denote the index of type K as $index(\mathbb{T}, K)$ the type under index i as $type(\mathbb{T}, i)$. Furthermore, we made an assumption that a single routine is unlikely to reference more than 64 distinct types. From this follows that we can represent a subset of $\mathbb{T}$ as a 64 bit integer, where bit with index i indicates whether the type $type(\mathbb{T}, i)$ is present in the set or not. With this representation, we can implement very efficient set operations with bitwise operators.

However, this representation does not reflect the lattice we described earlier. We shuffle the type indexes to the left and reserve first few indexes from right for special bit flags:

- **any flag** if present, other bits should be ignored and the whole type information value is deemed as the lattice element *Any*.
- **object flag** if present any class types should be ignored.
- **type hint flag**
- **subclasses flag**

From the properties of bitwise or, it follows that a bitwise or (union) of two type information instances represented with this schema, will give us their lowest upper bound. Let us, for example, consider a union of two type information

instances where one has **any flag** set. The union will have also **any flag** set, which in the lattice corresponds to $Any \wedge x = Any$ for every x. The same behaviour works with respect to the other special flags.

Last thing we need to deal with is when a routine references more types than the number we can represent. In such case, any type whose index would be greater than 63, is treated as if its index was exactly 63. This means that types with index greater or equal to 63 will share one bit and therefore we loose precision, because we cannot distinguish those types from each other. However, it is a safe approximation and a price we pay for the memory efficiency.

5. Results

We evaluated Control Flow for Phalanger on several open source PHP projects. In order to have a comparison to another established tool of similar type, we analysed one of the PHP projects with both Control Flow for Phalanger and Phantm. The discussion of the differences in results is provided in the following section. The rest of the PHP projects were analysed only with Control Flow for Phalanger and the reported problems were manually inspected and categorized.

5.1 Comparison to Phantm

The project we chose for comparison evaluation is Zebra_Image [21]. It is a PHP image manipulation library that consists of 1707 lines of code in one file and one class. Zebra_Image code contains some type documentation, but incomplete and often written in a way that does not follow the standard PHPDoc type description grammar.

All the problems reported by either tool were briefly inspected and categorized. The results are presented in the following table and are very different for each tool, which can be due to the different approaches to the analysis. The actual problems reported and the possible reasons for the difference in results are discussed in the following paragraphs.

Category	**Phantm**	**Control Flow for Phalanger**
All Reported problems	130	1
Uninitialized variable use	2	1
Zebra_Image dynamic fields	82	0
Any is not of type X	13	0
Prototype errors	9	0
Conversion: double to integer	2	0
Other type conversions	2	0
Not categorized	17	0

Reported Problems

The uninitialized variable error appears to be genuine problem that would generate a notice at runtime. It was discovered by both tools. Because the variable in question is accessed as an array using the `[]` operator, Phantm reported it twice.

All the prototype errors are false positives probably due to the fact that Phantm does not distinguish mandatory and optional parameters. In PHP, any routine can take any number of arguments. Arguments explicitly named in the signature are mandatory, but any number of optional arguments can be accessed using intrinsic functions. Therefore it is not necessarily a prototype error if a routine is given more parameters than the number of parameters explicitly named in its signature.

PHP allows to use undeclared object fields, they are dynamically created when first used, usually assigned to. Zebra_Image relies on this and dynamically

creates some fields in certain methods and then uses them in other methods, which is reported by Phantm. Using fields without explicitly declaring them can be seen as a style error, especially in case of Zebra_Image, because all the fields in question are in fact known beforehand and therefore it is perfectly possible to explicitly declare them. In other cases, however, this language feature enables useful patterns mainly in combination with magic `__set` and `__get` methods, as for example in model classes of Doctrine Object-Relational Mapper [22].

Phantm regards some implicit type conversions as errors. Namely anything to string type conversion, which is typically used in PHP, and double to integer, which is more debatable, but nonetheless often used deliberately instead of cast or floor or ceiling functions.

Conclusion

Those results can indicate that, although at first sight the aims of the two tools are the same, under closer inspection, they differ. Phantm attempts to perform very precise analysis. For example, by modelling heap memory. Such precision is, in theory, more powerful and can discover more actual problems; yet, it also yields many false positives. It would be possible to filter out the "likely to be" false positives, but that would filter out those cases where due to the better precision, Phantm discovered actual error.

On the other hand, although being less precise, Control Flow for Phalanger seems to have better ratio of false positives. This could make it more suitable option for a day to day development, which is what it was in fact focused on.

Due to this difference, the results for other PHP projects were manually inspected only in the case of Control Flow for Phalanger.

5.2 Evaluation on open source PHP code

For further evaluation, the following open source PHP frameworks and websites were chosen:

PHPUnit: a port of JUnit unit testing framework for PHP; it 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 has been used.

Nette: popular general purpose PHP framework with long development history and contributions from 137 developers. Nette has unit test suite with good code coverage and the source code is very well documented including the type documentation comments.

An evaluation always started with downloading a git repository with the latest source code of given PHP project. Then 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.

Often one real issue in the source code caused several warnings to be reported by the tool. For instance, if a documentation of a type of a field was not

correct, most the lines where a value was assigned to that field were reported, however, the root of the cause was actually the only one line with the wrong type documentation. Such cases were counted as a single problem.

5.2.1 Problems Taxonomy

The problems were divided into few main categories described below.

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.

5.2.2 Summary

The following table provides a summary of the problems found. There is a table that lists concrete problems found PHPUnit available in appendix.

Category	PHPUnit	Nette	Zebra_Image	total
Style	6	0	1	7
default return value	2	0	0	2
Documentation	10	1	0	11
missing false	3	0	0	3
Actual Error	1	0	0	1
False positive	8	0	0	8
unused arguments	1	0	0	1
amendable	4	0	0	4
built-in doc error	3	0	0	3
Total (excl. false positives)	17	1	1	19

5.2.3 Selected Problems

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.

6. Conclusion

In this thesis we presented a project with code name Control Flow for Phalanger, which can analyse PHP source code in order to discover type related errors and mismatches with type documentation.

The Control Flow for Phalanger was evaluated on three real world PHP project. Although, the tool does not use heap abstraction and does not perform context sensitive analysis, it was still capable to discovered several real issues with a good ratio of false positives.

This result may indicate that, were some imprecision in the analysis results can be tolerated, the modular approach we used can give results comparable to those of tools that use more complex methods.

6.1 Future Work

The Control Flow for Phalanger has not yet been fully integrated into the Phalanger project. This includes integration with the compiler in order to enable code optimizations and evaluation of the possible performance gain when running PHP websites like WordPress.

The analysis itself can be improved on several fronts.

Array support has not been implemented yet. Variables that can be of type array are analysed properly, but the structure of the array is not analysed, therefore any time an element of an array is accessed, we do not have any type information for it and have to assume the worst, that is it can be any type.

Type information is represented using a 64 bit value, however, we can go even further and represent the type information with 8 bit number, which will be an index into a table with all the possible type information instances for one routine. This would give us 255 possible combinations of types, which we assume is enough for most of the routines.

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

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

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

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

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

[9] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 194–206, ACM, 1973.

[10] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM, 1977.

[11] P. Cousot, "Abstract interpretation based formal methods and future challenges, invited paper," in *¡¡ Informatics — 10 Years Back, 10 Years Ahead ¿¿* (R. Wilhelm, ed.), vol. 2000 of *Lecture Notes in Computer Science*, pp. 138–156, Springer-Verlag, 2001.

[12] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *arXiv preprint arXiv:1403.4910*, 2014.

[13] K. Walker and R. E. Griswold, "Type inference in the icon programming language," *ACM Trans Prog. Lang. and Systems*, 1996.

[14] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, pp. 6–pp, IEEE, 2006.

[15] K. J. Hauzar D., "Weverca: Web applications verification for php (tool paper)," in *Accepted for publication in SEFM 2014, Grenoble, France. LNCS, September 2014*, 2014.

[16] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, *et al.*, "The hiphop compiler for php," in *ACM SIGPLAN Notices*, vol. 47, pp. 575–586, ACM, 2012.

[17] M. Madsen, P. Sørensen, and K. Kristensen, *Ecstatic–type inference for Ruby using the cartesian product algorithm*. PhD thesis, Master Thesis, Jun, 2007.

[18] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks, *Dynamic inference of static types for ruby*, vol. 46. ACM, 2011.

[19] E. Kneuss, P. Suter, and V. Kuncak, "Phantm: Php analyzer for type mismatch," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 373–374, ACM, 2010.

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

[21] "Zebra-image, a lightweight image manipulation library written in php." `http://stefangabos.ro/php-libraries/zebra-image/`. Accessed: 2014-07-05.

[22] "Object relational mapper – doctrine project." `http://www.doctrine-project.org/projects/orm.html`. Accessed: 2014-07-05.

List of Tables

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	
`Framework\Assert.php`	1861	Actual Error	Discussion in the thesis.
`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.

Attachments