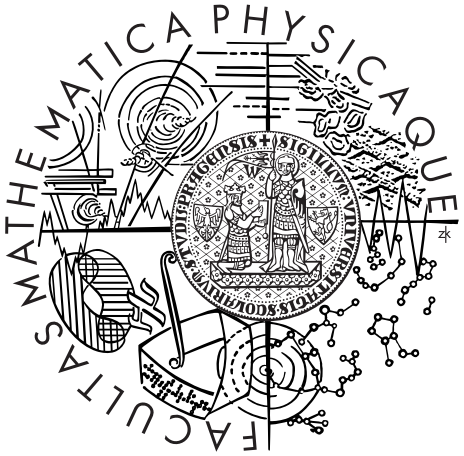
Charles University in Prague
Faculty of Mathematics and Physics

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:

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

Contents

1. Introduction

1.1 The Problem

Three out of top ten programming languages in TIOBE index[1] are dynamically typed languages. One of the reasons for their popularity is that they are usually 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 compilers or interpreters 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 document 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 an ongoing research of the static analysis methods for many different families of programming languages, including dynamic languages. The problem this project is addressing is to adapt and apply those methods on a real world and widely used programming language PHP. The result is a library that is capable of performing a static analysis of PHP code and can be integrated into the Phalanger project. The library should allow to plug in any kind of analysis, for example constant propagation. However, the main goal is to provide a type analysis in order to discover possible type related errors and mismatches with the type information in the documentation comments. Furthermore, in the future the library can be integrated into the Phalanger as a middle-end to provide optimizations for the compiler.

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

1.2 Thesis structure

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

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

2. Analysing PHP Code

The PHP programming language first appeared in 1995[3]. Over the years the language has involved and so have the ways how programmers were using 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 manner, using modern PHP patterns and idioms that are described later in this text. The analysis, however, should work reasonably good on any valid PHP code. We do not focus only on websites, but also on PHP libraries and frameworks that by themselves do not contain any PHP files that produce HTML or any other output for the user.

2.1 PHP semantics

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

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

2.1.1 Local Variables

Local variables in PHP do not need to be declared explicitly. Instead the first usage of a variable is also its declaration. If a variable's value is used before the variable got any value assigned, then the interpreter generates a notice, however the execution continues and value `null` is used instead. A variable can get a value assigned to it when it appears on a left hand side of an assignment or when a reference to that variable is created, in which case it gets value `null`. Note: references are discussed in one of the following subsections.

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

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

2.1.2 Global and Local Scope

2.1.3 Closures

PHP also supports anonymous functions. An anonymous function has its own scope as any other function and its local variables are not visible to the scope where it was declared

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

2.1.4 Interesting Control Flow Structures

Arbitrary expressions in continue and break.

2.1.5 Conditional Declarations

2.1.6 Auto-loading

2.1.7 PHPDoc Annotations

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 have to have a web server for example in order to analyse code of a web application. We can also guarantee some properties 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 us some results.

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

2.2.1 Terminology

Static analyses are usually described in the form of inference rules. An example of an inference rule can be "if the types of expressions e_1 and e_2 are integers, then the type of expression $e_1 + e_2$ is integer". Those rules can be more formally described with the following notation

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

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

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

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

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

```
if (...) $x = 3;
else $x = 'string';
$e_1 = $x;
$e_2 = 4;
$e_1 + $e_2;
```

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

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

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

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

and rewrite the inference rules as follows:

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

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

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

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

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

Then the rule will be:

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

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

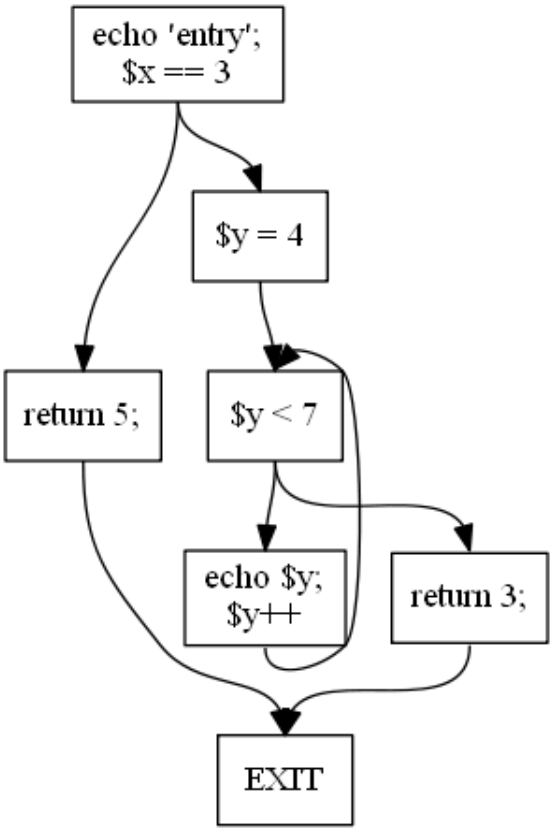
$$\frac{\vdash v_1 : \ominus \quad \vdash v_2 : \top}{\vdash v_1 + v_2 : \top}$$

2.2.2 Data Flow Analysis

Data Flow Analysis (DFA) is a static analysis framework for compiler optimizations and verification that scales to large code bases [4], [5].

Control Flow Graph. DFA is typically performed on a control flow graph, although there are also existing approaches to DFA without explicit control flow graph construction [6].

Control flow graph nodes, also called basic blocks, are program statements that are always sequentially executed. Directed edges represent the control flow between basic blocks, i.e. 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. 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.




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

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

return 3;
```

Table 2.1: Control flow graph

Lattices. The reasoning behind correctness and termination of DFA is based on an algebraic structures called lattices. A lattice is a partially ordered set in

which every two elements have a least upper bound, called supremum, and a greatest lower bound, called infimum.

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

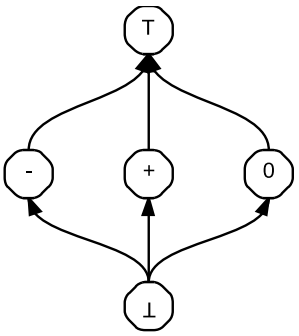


Figure 2.1: Bounded lattice with 5 elements.

What is important about lattices for DFA analysis is that if we have a lattice $(S, \leq_s)$ and a function $f : S \to S$ that is monotonous, i.e. $\forall a \in S : a \leq_s f(a)$, then f has a fixpoint. That is a point $x \in S$ for which $f(x) = x$.

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 an element that is greater than y, but this cannot go on forever, because eventually f will be given $\top$ for which it does not have any other option but returning $\top$ and we have a fixpoint again.

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

2.2.3 Intraprocedural Analysis

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

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

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

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

2.3 Control Flow for Phalanger Approach

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

3. Existing Software

3.1 Phantm[7]

3.2 HipHop Type Inference for Hack

3.3 Weverca: Web Verification Tool[8]

4. Implementation

4.1 Implementation Specific Constraints

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

Abstract Syntax Tree. Phalanger front-end parses PHP code into an Abstract Syntax Tree (AST) [5] structure. This structure is then traversed by the back-end using the visitor design pattern [9]. Phalanger does not use any other intermediate representation than AST.

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

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

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

The longer term aim of this project, not in the scope of this thesis, is to replace the existing algorithms for code completion, "jump to definition" and "find usages" features. Because with a dynamic language like PHP, it is not trivial to find all the usages of e.g. a class or determine a definition of e.g. 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 can be divided into several parts:

- Control Flow Graph construction,
- Data Flow Analysis framework,
- Code Tables maintenance,
- The analyses:
  - Dead Code Elimination,
  - Aliasing Analysis,
  - Constant Propagation,
  - Type Analysis.

The Data Flow Analysis is performed on a Control Flow Graph. However, the Data Flow Analysis is an abstract algorithm and in order to implement a concrete Data Flow Analysis, one must provide implementation of required missing operations.

An analysis is a concrete Data Flow Analysis or any other analysis, because the analysis can be performed, for example, only on the Control Flow Graph (Dead Code Elimination) or only on the function body (Aliasing Analysis) without using the Data Flow Analysis algorithm. The results of an analysis can be annotations added into the AST node objects, or annotations of the Control Flow Graph nodes, which support extensible attributes the way AST nodes do.

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 others. This information is mainly type related information, for example, a return type of a function. Code Tables can be also queried by an analysis, for example, when the Type Analysis reaches a function call and needs to find out what types the given function may return.

The whole project is designed as a class library with many extension points where some of the functionality can be used independently. However, in order to provide better usability, it 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 Control Flow Graph

Description of the Control Flow graph construction. Discussion of the choice of intermediate representation: in fact, why we do not use any intermediate representation, because we wanted to keep simple correspondence between the AST nodes and analyses results and because the back-end emits MSIL code from AST nodes.

4.4 Data Flow Analysis

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

4.5 Tables

Intra-procedural dependencies handling.

4.6 Aliasing and Constant Propagation Analysis

4.7 Type Analysis

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

4.8 The Analyser pipeline and AnalysisDriver

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

5. Results

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

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

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

5.1 PhpUnit

5.2 Zend Framework

5.3 Nette

5.4 WordPress

5.5 Drupal

6. Conslusion

6.1 Future Work

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

Bibliography

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

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

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

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

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

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

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

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

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

List of Tables

Attachments