Type inference for PHP
The value of annotations in a dynamic language

Ruud van der Weijde

March 20, 2015, 37 pages

Supervisor: Jurgen Vinju
Host organisation: Werkspot, http://www.werkspot.nl
Host supervisor: Winfred Peereboom

WERKSPOT
KLUS • KLIK • KLAAR
HEERENGRACHT 496, AMSTERDAM
http://www.werkspot.nl

UNIVERSITEIT VAN AMSTERDAM
FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA
MASTER SOFTWARE ENGINEERING
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Dynamic language are generally hard to statically analyse because of run-time dependencies. Without running the program there are many things unknown. Because dynamic languages are PHP are widely used, the need for decent analysis tool grows. This research examines the value of adding annotations to PHP code to improve the analysability. In the results we see that annotations improve the analysability of software code (this is a guess). Here I should state something about the correctness of the annotations. And end with a general conclusion.

Preface

In this section I will thank everyone who has helped me. Maybe also introduce some anecdote on how this research came to be.

Chapter 1

Introduction

1.1 PHP

PHP[1]is a server-side scripting language created by Rasmus Lerdorf in 1995. The original name 'Personal Home Page' changed to 'PHP: Hypertext Preprocessor' in 1998. PHP source files are executed using the PHP Interpreter. The language is dynamically typed, which means that the types of variables are examined during run-time. In statically typed languages all variable types are known at compile time. PHP supports duck-typing, which allows variables to change types during execution.

Evolution The programming language PHP evolved after its creation in 1995. In the year 2000 Object-Oriented (OO) language structures were added to the langue with the release of PHP 4.0. The 5th version of PHP was released in 2004 and provided an improved OO structure. Namespace were added in PHP 5.3 in 2009, to be able to resolve class naming conflicts between library and create better readable class names. Namespaces are comparable to packages in Java. OPcache extension is added was added in PHP 5.5 and speeds up the performance of including files on run-time by storing precompiled script byte-code in shared memory. The most recent stable version is 5.6 and includes more internal performance optimisations and introduces a new debugger.

Popularity According to the Tiobe Index[2]of December 2014, PHP is the 6th most popular language of all programming languages. The language has been in the top 10 since it's introduction in the Tiobe Index in 2001. More than 80 precent of the websites have a php backend[3]. The majority of these websites use PHP version 5, rather than version 4 or older versions. It is therefor wise to focus on PHP version from 5 and discard the older unused versions.

Analysability Although the popularity for more than a decade, there is still a lack of good PHP code analysis tools. Tools can help to reveal security vulnerabilities or find vulnerabilities or bugs in source code. The tools can also provide code completions or do automatic transformations which can be used to execute refactoring patterns. Source code analysis can be performed statically or dynamically or a combination of the two. More information on the analysability of php can be found in section 2.1.

1.2 Position

In this research we investigate how we can improve the analysability of PHP programs. We will show that the use of annotated source code can help to improve the analysability. The correctness of the annotations can also be examined by checking the implementation of the code.
These annotations can help to improve the analysis. The results can be used to find security issues, and if they are highly reliable we can even make compiler optimisations.

---

[1]http://php.net
[2]http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, December 2014
[3]http://w3techs.com/technologies/details/pl-php/all/all, December 2014

As far as we know, there is no constraint based type inference research like this one performed for PHP. That makes this research unique. There have been similar analysis for other dynamic languages, like smalltalk, ruby and javascript, but none like this.

1.3 Contribution

TODO Review this part when the result of the analysis are performed.

- Created an M3 for php.

- Constraint system.

- Show the value of annotations for analysis.

Some idea's are that this analysis can help IDE tools to perform transformations on the source code. (But the performance may not be sufficient.)
The creation of the M3 model can help to compare researchers compare PHP programs with other programming languages. For now only Java is implemented, but more can follow (unchecked statement).

1.4 Plan

The rest of this thesis is as follows: chapter 2 contains background and related work. Background exists of important language constructs, information about annotations in PHP, introduction to M^3 and type systems. The last section of this chapter shows similar research and their relation to this research. Chapter 3, research method, defines the research question and the research context. Chapter 4 describes the performed actions of this research. The analysis is presented in chapter 5, with the results in chapter 6. This thesis ends with the conclusions in chapter 8.

Chapter 2

Background and related work

Chapter 2 describes seven important language constructs which you need to understand in order to understand the difficulties of analysing PHP in section 2.1. The second section, section 2.2, gives an introduction to the syntax and usage of annotation in PHP. Section 2.3 explains the programming language Rascal which is used for analysis. M^3 is a programming language independent meta model which holds various facts about programs. M^3 is described in more details in section 2.4. Section 2.5 gives an introduction to type systems and how they are related to this research. The final section of this chapter, section 2.6, describes the related work and how these researches are related to this thesis.

2.1 PHP Language Constructs

This section presents for this research important language constructs and why these constructs are important. Explanations of these constructs help you to understand the performed analysis. The discussed parts are scoping, includes, conditional classes and functions, dynamic features, late static binding, magic methods and dynamic class properties.

Scoping   In PHP, all classes and functions are globally accessible once they are declared. All classes and functions are implicitly public, inner classes are not allowed, and conditional functions (see paragraph about conditional classes and functions) will be available in the global scope. If a class or function is declared inside a namespace, their name is prefixed with the name of the namespace.
Variables have three scope levels: global-, function-, and method-scope. Under normal circumstances when a variable is declared inside a function or method, their scope is limited to this function or method. Variables declared outside function or methods are available in the global scope, but not in the method or function scope. There is an exception for some predefined global variables which are available everywhere. Examples are `$GLOBALS`, `$_POST`, and `$_GET`. Variables inside a function or method can be aliased to a global variable by adding the keyword `GLOBAL` in front of the variable name. The variable are then linked to the global variable in the symbol table[1].
Closures (anonymous functions in PHP) have the same scoping rules as variables, but they can inherit variables from outside their scope by providing them in the use statement. In this research we will not support closures.

Includes   In PHP it is possible to include other PHP-files during execution of the program. These files will be loaded inline. This means that if you use an include in the middle of a file, the source code of this file will be inserted virtually at that place. In this research we will not perform an include analysis. Instead we will assume that all files in the project are included during execution.
According to the coding standard that is used in the php community[2], function- and class-name classes should not appear when using namespaces and autoloading.   When a class which is not loaded in memory is instantiated, the autoloading will try to include a file and load the class. The structure of the

---

[1] http://php.net/manual/en/language.variables.scope.php, July 2014
[2] http://www.php-fig.org/psr/psr-0/, July 2014

autoloading is meant to include classes, interfaces, traits and functions and should not have inline code executions which would lead to side-effects.

Conditional classes and functions Once a file is included in the execution, al the classes and functions in the top scope are declared. All class and function declarations within condition statements or within a method or function scope are only declared when the code is executed.

An example of an conditional statement can be found in listing 2.1. If the class Foo or function bar to not exist before the statements is executed, then the class and function will not yet declared. When you try to use the class or function, the script will die with an fatal error (if the class or function was not defined before).

```php
if (!class_exists("Foo"))
    class Foo { /* ... */ }

if (!function_exists("bar"))
    function bar() { /* ... */ }
```

Listing 2.1: Conditional class and function definitions

Listing 2.2 shows when functions and classes will be available. If the first call is g() as you can see in line 7, the script will result in a fatal error. When function f is executed, function g will be declared, but not yet class C. The class C will be declared once function g is executed. Once the functions and classes are declared, they are available in the top scope, possibly prefixed with the name of the namespace.

```php
function f() {
    function g() {
        class C {}
    }
}

g(); f(); // will fail because 'g();' is not declared yet
f(); g(); // will work because 'g();' is declared when calling 'f();'
f(); new C(); // will fail because 'g();' needs to be called first
f(); g(); new C(); // will work because 'g();' is called and has declared 'f();'
```

Listing 2.2: Conditional function declaration

Dynamic features PHP includes some dynamic features like: include dynamic variables, dynamic class instantiations, dynamic function calls, dynamic function creation, reflection, and eval. A previous study by Mark Hills[HKV13] has shown that the dynamic features are not used too much. (please double check this!!) Our focus will not be on trying to analyse these features, because we would need constant propagation. The downside of these dynamic features is that it will probably lower our precision. (please check this, and move this to another section)

Late static binding Late static binding[3]is implemented in PHP since version 5.3 by adding the keyword static to the language. It is similar to the keyword self, but it does not refer to the class it is declared in. The main difference is that self can be resolved statically, because it refers to the class it is declared in. static can only be resolved on runtime and represents the exact class that is instantiated.

Magic methods In PHP it is allowed to call methods or use properties that do not exists. Normally this would result in a fatal error, but not with the use of magic methods. One of the magic methods is het constructor method __call. This method is called when a non-accessible or non-existing method is called.

---

[3]http://php.net/manual/en/language.oop5.late-static-bindings.php, July 2014

Dynamic class properties Although it is a good practice to define your class properties, it is not required. On runtime it is possible to add properties to classes, even without the implementation of magic methods. In listing 2.3 you can see a code sample of adding a property after instantiation of a class. The acces of the non-existing propery `nonExistingProperty` will result in a warning, but code execution will continue and will just return `NULL`. The code on `line 5` is where the property is written. The object `$c` will have the `nonExistingProperty` publicly available now. But in a new class instantiation, like you can see on `line 6`, will not have the property there.

```php
class C {}
$c = new C();
var_dump($c->nonExistingProperty); // NULL
$p = $c->nonExistingProperty = "property now exists";
var_dump($p); // string(19) "property now exists"
$d = new C;
var_dump($d->nonExistingProperty); // NULL
```

<div align="center">Listing 2.3: Dynamic class property</div>

2.2 Annotations

PHP has no native support for annotations. But PHP has a `getDocComment`[4] method in the `ReflectionClass`. This `getDocComment` method returns the complete doc block of a certain element as a string. A doc block in php has the format `/**...*/`. Listing 2.4 shows an example of a doc block in PHP.

```php
namespace Thesis;

/**
 * Class Example
 * @package Thesis
 */
class Example
{
    /**
     * This is a description of the method getSomething
     *
     * @param SomeTypeHint $someTypeHint
     * @return bool
     * @throws CustomException
     */
    public function getSomething(SomeTypeHint $someTypeHint)
    {
        if ($someTypeHint->getType() == 'something') {
            throw new CustomException(sprintf('%s is not supported', $someTypeHint
                ->getType()));
        }

        return true;
    }
}
```

<div align="center">Listing 2.4: PHP DocBlock</div>

In PHP Annotations are not part of the official language. They are however widely used. For instance in ZEND, Symfony and Doctrine you can write business logic rules in the form of annotations. These annotations will be parsed and used in real code.

Other annotations are placed on top of classes, methods, functions, and variables. These annotations will help the developers to better understand what the code does. For example you can see what kind of input and output is expected for a method. IDE's will also use this information to better analyse the

---

[4]http://php.net/manual/en/reflectionclass.getdoccomment.php

source code.

Writing annotations is not yet in the PSR standards for PHP, but there is a proposal[5]. For this research we will only focus on the `@param`, `@return`, `@var`, and `@inheritDoc` annotations. The annotations `@return` and `@param` are only useful for functions and class methods. Type hints are described with `@var` and can be used on all structures, but mainly occur on variables and class fields.

There is no official standard for the use of annotations, but most projects follow the phpDocumentor syntax. For this research the following annotations are parsed:

$$\texttt{@return} = \begin{cases} \texttt{@return } type, & \text{unconditionally read } \texttt{@return } type. \end{cases}$$

(2.1)

$$\texttt{@param} = \begin{cases} \texttt{@param } type \ \$var, & \text{if `}\texttt{@param } type \ \$var\text{' occurs at least once.} \\ \texttt{@param } \$var \ type, & \text{else if `}\texttt{@param } \$var \ type\text{' occurs at least once.} \\ \texttt{@param } type, & \text{otherwise try to match `}\texttt{@param } type\text{'.} \end{cases}$$

(2.2)

$$\texttt{@var} = \begin{cases} \texttt{@var } type \ \$var, & \text{if `}\texttt{@var } type \ \$var\text{' occurs at least once.} \\ \texttt{@var } \$var \ type, & \text{else if `}\texttt{@var } \$var \ type\text{' occurs at least once.} \\ \texttt{@var } type, & \text{otherwise try to match `}\texttt{@var } type\text{'.} \end{cases}$$

(2.3)

$$type = \begin{cases} type|type, & \text{if `|' in } type. \\ type, & \text{otherwise} \end{cases}$$

(2.4)

2.3   Rascal

Rascal is a meta programming language developed by the Centrum Wiskunde & Informatica (CWI)[KSV09]. Rascal is designed to analyse, transform and visualise source code. The language is build on top of Java and implements various constructs of existing programming languages. In this research, most of our code is implement in rascal.

2.4 M^3

The M^3-model is a model which holds information of source code[Izm+13]. This model is created to analyse one single Java program or compare two or more Java systems with each other. The core of the M^3-model contains `containment`, `declarations`, `documentation`, `modifiers`, `names`, `types`, `uses`, `messages`.

The `declarations` relation contains class, method, variable- information with their logical name and their real location. The type of the relation are `locations` and represent the logical name of the declaration and will be used in the rest of the M^3. The `containment` relation has information on what declarations are contained in each other. For example a package can contain a class; a class can contain fields and methods or an inner class; a method can contain variables. The `documentation` relation contains all comments from the source code. The `modifiers` relation has information on the modifiers of declarations. Modifiers are abstract, final, public, protected or private. The `names` relation contains a simplified name of the full declarations. The `types` relation has information about the type of the source code elements. The `uses` relation describes what reference is using which object. For instance when a field of a class is used in some expression, the `uses` relation links the field in the expression to the declaration of the field in the class. And lastly, `messages` contains errors, warnings or info statements.

---

[5]https://github.com/php-fig/fig-standards/pull/169/files, July 2014

2.5   Type system

The type system defines how rules are applied to types. The system determines validates the type usage with type checking. The process of actually resolving types is called type inference. On one hand the system needs to determine the type of variables, on the other hand it will check the type of the variables.

Type checking   Type checking of dynamic languages differs from static languages. In dynamic languages, most type checking is performed during run-time. The down side here is that you actually have to run the program to get feedback on incorrect usage of types. For example when you divide an integer by a boolean with value true. In PHP these operations are (most of the times) valid and will return you a value. In this case, the boolean will be internally concerted to the integer 1. This can result in unexpected behaviour and is discouraged to do.

On the other hand we have static type systems which allow type checking without executing the code. Most analysis can be performed during compile time. These systems can catch programming errors related to typing issues and can avoid overhead of runtime type checks. In dynamic languages the type checking is done during run time, and this will decrease the performance.

Type inference   In the previous section we explained the type checking. In order to check the types of variables and expression, we need to know the types of the variables and expressions first. The process of resolving the type of a variable or expressions is called type inference. In dynamic languages like PHP it can be difficult to resolve the type of a variable or expression without running the program. When you are able to resolve the type before running the program, you are able to optimise the execution code. This allows you to make performance improvements or avoid memory usage.

In this research we use constraint based type inference. Every tapeable language construct will have a set of possible types. At the start of the analysis everything will have all possible types, we call this `Universe`. Then we will read all the constraints and will try to solve them. Every time a constraint is solved, the number of possible types for one variable will decrease. This process will be repeated until we have solved all the constraints and can no longer limit the number of types for a variable.

Other forms of type inference are k-CFA (k-control flow analysis) and CPA (cartesian product algorithm). The analysis we perform is flow insensitive, which means that we do not analyse the flows of the program. We do take the context of the programs into account, making the analysis context sensitive.

(todo, describe K-CFA and CPA in more details)

2.6   Related work

Describe these:

- 'The HipHop Compiler for PHP'[Zha+12] (not much information available on their type inference, only source code)

- 'Phantm: PHP analyzer for type mismatch'[KSK10a; KSK10b] (investigate this in more details, their focus is on finding type errors)

- PHPLint [6](uses a different kind of annotations, not the Java like PhpDocs)

- 'Soft typing and analyses on PHP programs'[], code implementations: https://github.com/henkerik/typing and https://github.com/marcelosousa/soft-typing-PHP5 (created for php4, code for php5, should check this out, might be able to compare results with this)

- 'Design and Implementation of an Ahead-of-Time Compiler for PHP'[Big10] (to check in detail)

Also describe their differences with my research.

---

[6]http://www.icosaedro.it/phplint, July 2014

Chapter 3

Research Method

In this research we will try to answer research questions. Section 3.1 describes the research questions. In section 3.2 we will explain more about types, subtypes and their relationship. Section 3.3 explains in which context the research is executed.

3.1 Research question

The main research question we want to answer is:

Will the use of annotations[1] help to do better static source code analysis?

Subquestions:

- What is the accuracy (recall and precision) of our analysis?
- Are annotations used in programs reliable?

The subquestions are here to support the main question. In the first question we want to measure the precision and recall of this research. With precision we want to measure if the results we have are reliable. With recall we want to have the number of results we were able to reveal.

In the second subquestion we want to know if the annotations conform to the documentation. We will measure this by doing the analysis without using annotations, and see if the results fit the annotations.

3.2 Types

PHP is dynamically typed and allow coercions and duck-typing. Coercion make it possible for variables to hold different types Duck-typing checks the object instead of the class.

```
module lang::php::m3::TypeSymbol

data TypeSymbol
  = \any()         | array(TypeSymbol arrayType)
  | boolean()      | class(loc decl)
  | float()        | integer()
  | null()         | object()
  | resource()     | string()
  ;
```

---

[1]The annotations are limited to: @param, @return, @var, and @inheritDoc.

12

3.2.1 PHP types

PHP has a similar class inheritance structure and interface implementation as Java. The main difference is that in PHP all class are `public` and that inner classes are not allowed in PHP.

The basis types in PHP are integers, floats (similar to doubles and reals), booleans, strings, arrays, resources and null. When variables are initialised without a values, they are null. The recourse type is a special one which is not important for this research.

3.2.2 Subtypes

Explain something about subtypes here. For now, only this figure3.1
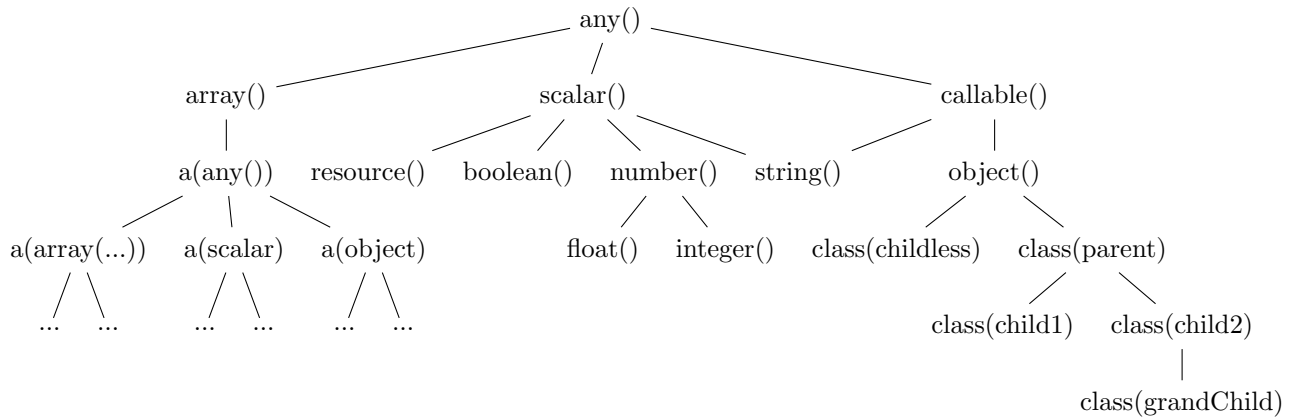
Figure 3.1: Subtype hierarchy

The subtype relation of class inheritance is a reflexive transitive closure relation. A class extension of `class A` on `class C` will define `class A` as a subtype of `class C` in our analysis, as you can see in figure 3.2. If a class does not extend another class, it will implicitly extend the stdClass class. You can see that this happens with `class D` in the example. The `stdClass` is represented as the type `object()` in our analysis.

The basic PHP types also contain a subtype relation. Integers are subtypes of floats.

(a) Inheritance relation   (b) Subtype relation   (c) Inheritance in PHP

Figure 3.2: Relation of subtypes among classes

3.3 Research context

In order to let our research take place, we need to make sure that some environment variables are constant.

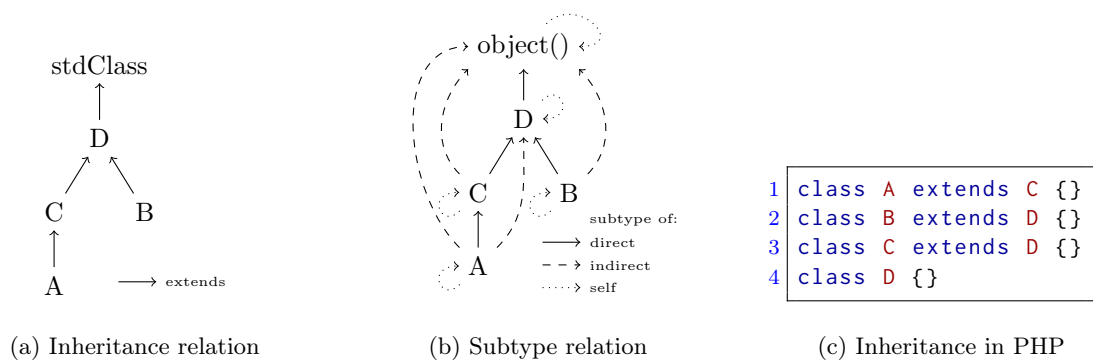**Program Correctness** In order to be able to execute this research we will need to assume that the program is correct and works as intended. We will assume that the system contains no bugs. This is needed to be able to say something about the programs we analyse.

File includes In this research we will assume that all file are included during runtime. When a PHP system is constructed of classes with namespaces, the files will be logically loaded using PHP's autoloader. Because most recent systems use namespaces, we will assume that all files are included. For legacy systems, this can influence the results of this research.

Register globals Register globals allows variables to be magically be created from GET and POST values. Since it is discouraged to use this setting, we will assume that all software products have this setting disabled.

PHP Warnings For this research we will ignore all warnings. Warnings do not alter the behaviour of the program. In a normal production environment, the warnings are not shown.

Sensitivity Our analysis is flow-, control-, and context-insensitive. Flow-insensitive means that we do not look at messages between objects, and only look in the body of a method. Control-insensitive means that we ignore all control structures. Examples of control structures are `if`, `else`, and `switch`. Context-insensitive means that we do not look at the order of which code is executed.

Chapter 4

Research

The research is executed in 3 main steps. The first step creates an M^3 model for a PHP program and contains various facts about the program. The creation of an M^3 for PHP is explained in more details in section 4.1. Once the M^3 model is constructed, the second step is to extract constraints from the program using the M^3 model. How and which constraints are extracted is described in section 4.2. In the third step, in section 4.3, the constraints are solved and resolve the types of variables used in the program. The final section 4.4 of this chapter explains how annotations can be included in the process to gain more precise results.

4.1 M^3 for PHP

As explained in section 2.4, M^3 is a language independent meta model which holds facts about programs. The model can be extended with language specific elements and will be used to query the system for facts about the system. An overview how an M^3 for PHP is build is shown in figure 4.1. Independent M^3 models are build each PHP file in the program. All these individual M^3's are in the end combined to collect all the facts about a program. This results in one M^3 for the whole program. M^3's are first created for each file is because it is not defined what the dependencies of a individual file are. There is no main file, and all files can load each other. In this research we assume that all files in a program are loaded when needed using autoloaders or manual includes.
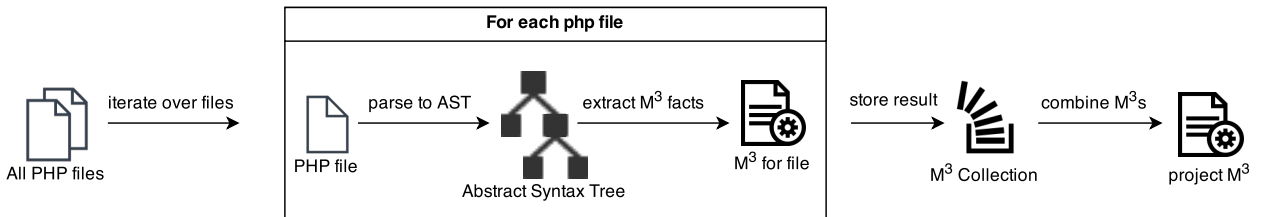
Figure 4.1: M^3 Creation

4.1.1 The algorithm

In order to get a better understanding of the creation of an M^3 for PHP, the algorithm is provided in Algorithm 1. **The input** of the algorithm is all the `PHP files of a program`, which are all files ending on `.php`. **The output** is an M^3 with facts about the provided program. The algorithm starts with initialising an empty M^3 collection in line 1 which with be filled with the result of each individual files. In the big loop on line 2 to 13 we iterate over all the PHP files of the program. The first thing that needs to be done is to create an Abstract Syntax Tree (AST) of the program using an external PHP parser[1] which returns an Rascal AST in `parseUsingPhpParserAndReturnRascalAST` on line 3. From this AST we create an

---

[1]https://github.com/ruudvanderweijde/PHP-Parser

empty M^3 in `createEmptyM3` on line 4. In order to be able to refer to any source code element, we need to have the declarations of the elements. These elements are extracted in `addDeclarations` on line 5 can be `namespace`, `class`, `interface`, `trait`, `method`, `function`, `variable`, `field`, or `constant`. For all `functions` we need to add scope information in case functions are declared inside another function or method. In line 6 we add the scope information to all nodes to the AST by visiting the AST. This is needed in order to extract the right facts about the logical containment which is done in line 7. The next three lines (8-10) extract facts about class inheritance and the interface implementations, modifiers, PHP doc blocks, and annotations. Once all the basic information is collected, `calculateUsesFlowInsensitive` on line 11 tries to find the declarations of the used objects, methods, functions and variables. This is flow and context insensitive and only tries to resolve non-dynamic language constructs. The last step of the iterator is to add the constructed M^3 to the M^3 collection. Finally when an M^3 is constructed for all files, all facts of the individual M^3's are merged into one M^3 model in line 14.

Algorithm 1: PHP program to M^3

Input: PHP files of a program
Output: M^3 for the PHP program

1 m3Collection = [];

2 forall the *file* ∈ *program* **do**

3   ast = parseUsingPhpParserAndReturnRascalAST(file);

4   m3 = createEmptyM3(ast);

5   m3 = addDeclarations(m3, ast);

6   ast = addScopeInformation(m3, ast);

7   m3 = addContainment(m3, ast);

8   m3 = addExtendsAndImplements(m3, ast);

9   m3 = addModifiers(m3, ast);

10   m3 = addRawDocBlocksAndAnnotations(m3, ast);

11   m3 = calculateUsesFlowInsensitive(m3, ast);

12   m3Collection += m3;

13 end

14 return projectM3 = composePhpM3(m3Collection);

4.1.2   Core elements

The M^3 model has the following core elements: `Declarations`, `Containment`, `Modifiers`, `Extends`, and `Uses`. All these core elements are explained in more detail in the paragraphs below.

`Declarations`   defines the declarations of namespaces, classes, interfaces, traits, methods, functions, and variables and holds the relation between the logical name (which is used to refer to the declaration) and the actual file location (which is the physical place in the file system. For example the logical name of a class can be `|php+class://SomeNameSpace/ClassX|` while the actual location might be `|file:///project/SomeNameSpace/ClassX.php|`.

`Containment`   holds information about what elements logically contain other elements. For example, a property or method is contained in a class and a class is contained in a package. When a function is declared in another function, they are both logically contained in the global namespace (the highers level) because all functions are declared as first class citizens in PHP.

`Modifiers`   element contains information about the modifiers of classes, fields, and methods. Classes can only be `abstract`, fields can be `public`, `private`, or `protected`, and methods can be all of them. Abstract methods can only be declared in abstract classes. Classes are implicitly public.

Extends contains information about what classes and interface extend other classes of interfaces. Please note that we do not hold information about which class implements which interface, because that information is contained in the `implements` relation. Interface extensions work just like class extensions.

Uses relation holds information about the usages of certain elements. It is the relation between the usage and the declaration. For example when you instantiate a class, in that case you 'use' that specific class.

4.1.3   PHP specific elements

Because every programming language differs in syntax and semantics, the M^3 model is extensible to provide language specific modeling elements. The following php specific items are added: `Implements`, `TraitUses`, `Parameters`, `Constructors`, `Aliases`, and `Annotations`. These PHP specific modeling elements are described in more detail in the paragraphs below.

Implements holds information on what class implements which interfaces. One class can implement multiple interfaces. Because the information is a relation between the class and the interface, we can easily add multiple interfaces to one class in the model.

TraitUses knows which traits a class uses. Traits are a collection of reusable functions, are defined on class level, and one class can have multiple trait usages. All the methods of a trait are on runtime imported in the class.

Parameters keeps track of the parameters of a method or function. This information is stored to make it easier to resolve the call to a method or function.

Constructors lists all the constructors for classes. This information is needed because it is not always clear what constructor is used, due to legacy PHP4 way of using class constructors. In PHP4 the constructor was defined as a method with the same name as the class. Since PHP5 the language is provided with a magic method `__construct()`, which results in two ways to have constructors, but only one constructor will be called. The PHP4 constructors will be removed in PHP7.

Aliases has a relation between aliases and the actual implementation. For instance the function `class_alias` defines a new name for the same class. This relation is also used to keep track of references.

Annotations contain a relation between a declaration and the annotations that are known. Annotations are defined in the raw doc blocks and are parsed using regex. Regex was in this case easier then parsing because there is no official grammar or standard defined. For this research we only use `@param`, `@var`, `@returns`.

4.2   Fact extraction

We can extract fact about classes, class-constants/fields/methods, functions, parameters. For these facts, we can use a relation, so we have a many-to-many relation. On the left size we will have the class, function or method. On the right side we have their attribute.

Other facts that will be used:

```
alias PhpParams = lrel[loc decl, set[loc] typeHints, bool isRequired, bool byRef];
data Annotation = returnType(set[TypeSymbol]) | parameterType(loc var, set[TypeSymbol])
    | varType(loc var, set[TypeSymbol]);

anno rel[loc from, loc to] M3@containment;          // 'from' directly contains 'to'
```

```
anno rel[loc from, loc to] M3@extends;              // 'from' extends 'to'
anno rel[loc from, loc to] M3@implements;           // 'from' implements 'to'
anno rel[loc decl, PhpParams params] M3@parameters; // formal parameters of functions/methods
anno rel[loc decl, loc to] M3@constructors;         // 'decl' and its constructor 'to'
anno rel[loc decl, Annotation annotation] M3@annotations;  // result of parsed php docs
```

4.2.1 Type extraction

In order to define the subtype relations in class extensions, we will need to declare all existing class types.
We can do this in rascal like is done in the example below:

```
visit (system) {
    case c:class(_, _, _, _, _):  types += class(c@decl);
}
```
Once all types are defined, we can add the subtype relation. We will need to have the subtype of

`int()` and `float()` and the class extensions. You can see that in the code below:

```
public rel[TypeSymbol, TypeSymbol] getSubTypes(M3 m3, System system)
{
    rel[TypeSymbol, TypeSymbol] subtypes
        // add int() as subtype of float()
        = { <\int(), float()> }
        // use the extends relation from M3
        + { <class(c), class(e)> | <c,e> <- m3@extends }
        // add subtype of object for all classes which do not extends a class
        + { <class(c@decl), object()> | l <- system, /c:class(n,_,noName(),_,_) <- system[l]
};

    // compute reflexive transitive closure and return the result
    return subtypes*;
}
```

4.2.2 Constraint extraction

(based on these rules, we can add constraints to the source code)
Introduction is needed here... for now I will just list the types that I have found. Maybe this needs to
be moved to a different chapter.
This is a list of items which are not supported (yet):

- References (in PHP they are symbol table aliases)

    - on expression assignments :: $a = \&$b$
    - on functions :: function $\&f()$ {...}
    - on parameters :: function $f(\&$a)$ {...}

- Variable structures:

    - ~~Variable variables~~ :: $$a;
    - ~~Variable class instantiation~~ :: new $a;
    - ~~Variable method or function calls~~ :: $a();

- List assign :: $list($a,$b) = array("one","two")$; (we can assume that the rhs is of type array,
  when the program is correct)

- ~~Method or function parameters (including type hints)~~

- ~~Class structures, method calls~~

- ~~Class Constants~~

- ~~The global statement~~ (should be resolved by the usage relation from M3)

- ~~Casts of expressions~~

- Parameters

- ~~Predefined variables~~ ($this, self, parent, static)

- ~~Eval~~ (will not be supported)

- ~~Closures~~ (not used much in production code)

- ~~Traits~~ (not used much in production code)

- ~~Callable~~ (introduced in 5.4 as typehint, not used much in production code)

- Foreach($a as ... (=> ...)) => $a is an array or an object;

- ~~return; => return type is null~~ (is added to the situation when there are no return statements)

- add predefined globals (and their type: $[GLOBALS, _SERVER, _GET, _POST, _REQUEST, _COOKIE, _ENV, _SESSION, php_errormsg] (all in global scope))

- add magic constants: _ _[DIR, FILE, LINE, NAMESPACE, FUNCTION, CLASS, METHOD]_ _

- ~~predefined constants: TRUE(b), FALSE(b), NAN(f), INF(f), NULL(n), STDIN(r), STDOUT(r), STDERR(r)~~

- define("name", value) mixed with constants (?out of scope?)

- ~~keywords: self, parent, static in a class~~ (is included in method and property calls)

- ADD CONSTANTS! RECORD THE TYPE OF THE DEFINED CONSTANTS AND TRY TO READ THEIR TYPE.

Legend

$=$	$=$	Equal to (type)	C	$=$	A class
$<:$	$=$	Is subTypeOf	$\to c$	$=$	A class constant
E_k	$=$	An expression	$\to p$	$=$	A class property
$[E_k]$	$=$	Type of some expression	$\to m$	$=$	A class method
f	$=$	A function	$[m]$	$=$	(Return) type of a method call
$[f]$	$=$	(Return) type of a function	(A_n)	$=$	The n'th actual argument
$:: c$	$=$	Static property fetch	(P_n)	$=$	The n'th formal parameter
$:: m$	$=$	Static method call	th	$=$	Type hint
$:: p$	$=$	Static property fetch	v	$=$	Default value
Mfs	$=$	Modifiers	Γ	$=$	Whole program

Table 4.1: Constraint legend

A list of predefined items can be found here:

- Constants

- Variables

- todo: functions

- todo: classes

Expressions

Normal assignment

$$\frac{E \equiv (E_1 = E_2)}{[E_2] <: [E_1],}$$
$$[E_1] <: [E]$$

```
1  $a = $b; // [$b] <: [$a], [$a] <: [$a=$b]
2  $c = $d = $e; // [$e] <: [$d], [$d] <: [$c],
3                // [$d] <: [$d=$e], [$c] <: [$c=$d=$e]
```

Listing 4.1: Normal assignment

Ternary

$$\frac{E \equiv (E_1 \; ? \; E_2 : E_3)}{[E] <: [E_2] \vee [E] <: [E_3]}$$

$$\frac{E \equiv (E_1 \; ? \; : E_3)}{[E] <: [E_1] \vee [E] <: [E_3]}$$

```
1  $expr ? $b : $c; // typeOf(E) is subtypeOf($b) or subtypeOf($c)
2  $expr ? : $c; // typeOf(E) is subtypeOf($expr) or subtypeOf($c)
```

Listing 4.2: Ternary

Assignments with operators (1) always resulting in ints

$$E_1 \; \& = E_2$$
$$E_1 \mid = E_2$$
$$E_1 \; \hat{} = E_2$$
$$E_1 <<= E_2$$
$$E_1 >>= E_2$$
$$\frac{E_1 \; \% = E_2}{[E_1] = integer()}$$

```
1  $a &= $b;   /* $a = integer() */
2  $a |= $b;   /* $a = integer() */
3  $a ^= $b;   /* $a = integer() */
4  $a <<= $b; /* $a = integer() */
5  $a >>= $b; /* $a = integer() */
6  $a %= $b;   /* $a = integer() */
```

Listing 4.3: Assignments with operators (1)

Assignments with operators (2) string concat (. =)

$$\frac{E_1 \; .= E_2}{[E_1] = string(),}$$
$$if([E_2] <: object()) => hasMethod([E_2], "\_\_tostring")$$

```
1  $a .= $b;   /* $a = string() */
2  // An error occurs when $b is of type object() and
3  // __toString is not defined or does not return a string
```

Listing 4.4: Assignments with operators (2)

Assignments with operators (3) resulting in int where rhs is no array

$$\frac{\begin{array}{c} E_1 \;/= E_2 \\ E_1 -= E_2 \end{array}}{\begin{array}{c} [E_1] = integer(), \\ [E_2] \neq array(any()) \end{array}}$$

```
1 $a /= $b;  /* $a = integer() */
2 $a -= $b;  /* $a = integer() */
3 // An error occurs when $b is of type array() for /= and -=
4 // Fatal error: Unsupported operand types
```

Listing 4.5: Assignments with operators (3)

Assignments with operators (4) resulting in int or float

$$\frac{\begin{array}{c} E_1 \;*= E_2 \\ E_1 += E_2 \end{array}}{[E_1] <: float()}$$

```
1 $a *= $b; /* when $b == (boolean()|integer()|null()) */ /* $a = integer() */
2 $a *= $b; /* when $b != (boolean()|integer()|null()) */ /* $a = float() */
3 $a += $b; /* when $b == (boolean()|integer()|null()) */ /* $a = integer() */
4 $a += $b; /* when $b != (boolean()|integer()|null()) */ /* $a = float() */
```

Listing 4.6: Assignments with operators (4)

Unary operators

$$\frac{E \equiv (+E_1) \vee (-E_1)}{\begin{array}{c} [E] <: float(), \\ [E_1] \neq array(\backslash any()) \end{array}}$$

$$\frac{E \equiv (!E_1)}{[E] = boolean()}$$

$$\frac{E \equiv (\sim E_1)}{\begin{array}{c} [E_1] = float() \vee [E_1] = integer() \vee [E_1] = string(), \\ [E] = integer() \vee [E] = string() \end{array}}$$

$$\frac{E \equiv (E_1 + +) \vee (E_1 - -)}{\begin{array}{c} if([E_1] <: array(\backslash any())) => [E] <: array(\backslash any()), \\ if([E_1] = boolean()) => [E] = boolean(), \\ if([E_1] = float()) => [E] = float(), \\ if([E_1] = integer()) => [E] = integer(), \\ if([E_1] = null()) => [E] = integer() \vee [E] = null, \\ if([E_1] <: object()) => [E] <: object(), \\ if([E_1] = resource()) => [E] = resource(), \\ if([E_1] = string()) => [E] = integer() \vee [E] = float() \vee [E] = string() \end{array}}$$

$$\frac{E \equiv (++E_1) \vee (--E_1)}{\begin{array}{c} if([E_1] <: array(\backslash any())) => [E] <: array(\backslash any()), \\ if([E_1] = boolean()) => [E] = boolean(), \\ if([E_1] = float()) => [E] = float(), \\ if([E_1] = integer()) => [E] = integer(), \\ if([E_1] = null()) => [E] = null, \\ if([E_1] <: object()) => [E] <: object(), \\ if([E_1] = resource()) => [E] = resource(), \\ if([E_1] = string()) => [E] = integer() \vee [E] = float() \vee [E] = string() \end{array}}$$

```
1  +$a // positive
2  -$a // negation
3  !$a // not
4  ~$a // bitwise not
5  $a++ // post increase
6  $a-- // post decrease
7  ++$a // pre increase
8  --$a // pre decrease
```

Listing 4.7: Unary operators

Binary operators

$$\frac{E \equiv (E_1 + E_2)}{\begin{array}{c} [E] <: array(\_) \vee [E] <: float(), \\ if([E_1] <: array(\_) \wedge [E_2] <: array(\_)) => [E] <: array(\_), \\ if([E_1]! <: array(\_) \vee [E_2]! <: array(\_)) => [E] <: float() \end{array}}$$

$$\frac{E \equiv (E_1 - E_2) \vee (E_1 * E_2) \vee (E_1/E_2)}{\begin{array}{c} [E] <: float(), \\ [E_1] \neq array(\_), \\ [E_2] \neq array(\_) \end{array}}$$

$$\frac{E \equiv (E_1 \ \% \ E_2) \vee (E_1 << E_2) \vee (E_1 >> E_2)}{[E] = integer()}$$

$$\frac{E \equiv (E_1 \ \& \ E_2) \vee (E_1 \mid E_2) \vee (E_1 \ \hat{} \ E_2)}{\begin{array}{c} [E] = string() \vee [E] = integer(), \\ if([E_1] = string() \wedge [E_2] = string()) => [E] = string(), \\ if([E_1] \neq string() \vee [E_2] \neq string()) => [E] = integer() \end{array}}$$

```
1   $a + $b // addition
2   $a - $b // subtraction
3   $a * $b // mulitiplication
4   $a / $b // division
5   $a % $b // modulus
6   $a & $b // bitwise And
7   $a | $b // bitwise Or
8   $a ^ $b // bitwise Xor
9   $a << $b // bitwise shift left
10  $a >> $b // bitwise shift right
```

Listing 4.8: Binary operators

Comparison operators

$$E \equiv (E_1 == E_2)$$
$$E \equiv (E_1 === E_2)$$
$$E \equiv (E_1 \; != \; E_2)$$
$$E \equiv (E_1 <> E_2)$$
$$E \equiv (E_1 \; ! == \; E_2)$$
$$E \equiv (E_1 < E_2)$$
$$E \equiv (E_1 > E_2)$$
$$E \equiv (E_1 <= E_2)$$
$$\frac{E \equiv (E_1 >= E_2)}{[E] = boolean()}$$

```
1  $a == $b   /* boolean() */
2  $a === $b  /* boolean() */
3  $a != $b   /* boolean() */
4  $a <> $b   /* boolean() */
5  $a !== $b  /* boolean() */
6  $a < $b    /* boolean() */
7  $a > $b    /* boolean() */
8  $a <= $b   /* boolean() */
9  $a >= $b   /* boolean() */
```

Listing 4.9: Comparison operators

Logical operators

$$E \equiv (E_1 \; and \; E_2)$$
$$E \equiv (E_1 \; or \; E_2)$$
$$E \equiv (E_1 \; xor \; E_2)$$
$$E \equiv (E_1 \; \&\& \; E_2)$$
$$\frac{E \equiv (E_1 \; || \; E_2)}{[E] = boolean()}$$

```
1  $a and $b  /* boolean() */
2  $a or $b   /* boolean() */
3  $a xor $b  /* boolean() */
4  $a && $b   /* boolean() */
5  $a || $b   /* boolean() */
```

Listing 4.10: Logical operators

Array

Array value fetch

$$\frac{E \equiv E_1[E_2]}{[E_1] \neq object(),}$$
$$if([E_1] = string()) => [E] = string(),$$
$$if([E_1] = array(\{types\})) => [E] <: \{types\},$$
$$if([E_1] \neq string() \land [E_1] \neq array(\_)) => [E] = null()$$

```
1  $a[0];
2  // typeOf($a) != object()
3  // when typeOf($a) == string() => typeOf($a[/*...*/]) is string()
4  // when typeOf($a) == array()  => typeOf($a[/*...*/]) is mixed()
5  // when typeOf($a) !== string|array => typeOf($a[/*...*/]) is null()
```

Listing 4.11: Array value fetch

Array declaration

$$\frac{E', \text{ where } E' \text{ is an array declaration}}{[E'] <: array(any())}$$

```
1  array(/*...*/); // typeOf() = array();
2  // Rascal: array(_) => array(\any())
```

Listing 4.12: Array declaration

Scalars

Scalars

$$\frac{E, E \text{ is a string}}{[E] = string()}$$

$$\frac{E, E \text{ is a float}}{[E] = float()}$$

$$\frac{E, E \text{ is a integer}}{[E] = integer()}$$

```
1  "Str" // string()
2  'abc' // string()
3  100 // integer()
4  1.4 // float()
```

Listing 4.13: Scalars

Encapsulated strings

$$\frac{E, E \text{ is an encapsed string*}}{[E] = string()}$$

* When a string contains expression(/variables), it is processed as encapsed.

```
1  "$var"
```

Listing 4.14: Encapsulated strings

Casts

Casts

Note: PHP Warnings are ignored

$$\frac{E \equiv (array)E_1}{[E] <: array(any())}$$

$$\frac{E \equiv (bool)E_1 \lor (boolean)E_1}{[E] = boolean()}$$

$$\frac{E \equiv (float)E_1 \lor (double)E_1 \lor (real)E_1}{[E] = float()}$$

$$\frac{E \equiv (int)E_1 \lor (integer)E_1}{[E] = integer()}$$

$$\frac{E \equiv (object)E_1}{[E] <: object()}$$

$$\frac{E \equiv (string)E_1}{[E] = string(),}$$
$$if([E_1] <: object()) => hasMethod([E_1], `\_\_tostring')$$

$$\frac{E \equiv (unset)E_1}{[E] = null()}$$

```
1 (array)$a  // <: array(\any())
2 (bool)$a    // boolean()
3 (float)$a   // float()
4 (int)$a     // integer()
5 (object)$a  // object()
6 (string)$a  // string(), when $a == object() the object needs to have __toString()
7 (unset)$a   // null()
```

Listing 4.15: Casts

Clone

Clone

$$\frac{E \equiv clone(E_1)}{[E] <: object(), [E_1] <: object()}$$

```
1 clone($a) // typeOf($a) = object, typeOf(clone($a)) = object
```

Listing 4.16: Clone

Class

Class instantiation (1) matching the class name

$$\frac{E \equiv new\ C_1()}{[E] = \text{class(C.decl)}}$$

```
1  new C;
```

Listing 4.17: Class instantiation (1)

Class instantiation (2) of an expression

$$\frac{E \equiv new\ E_1}{[E] <: object(),}$$
$$[E_1] <: object() \vee [E_1] = string()$$

```
1  $c = "C";
2  new $c;
```

Listing 4.18: Class instantiation (2)

Special keywords `self parent parent static`

$$\frac{E \equiv \$this \in C}{[E] <: \text{object()}}$$
$$[E] = \text{class(C)} \vee [E] :> \text{class(C)}$$

$$\frac{E \equiv self \in C}{[E] <: \text{object()}}$$
$$[self] = \text{class(C)}$$

$$\frac{E \equiv parent \in C}{[E] <: \text{object()}}$$
$$[E] :> \text{class(C)}$$

$$\frac{E \equiv static \in C}{[E] <: \text{object()}}$$
$$(\ [E] <: \text{class(C)} \vee [E] :> \text{class(C)}\ )$$

```
1  // $this can only be used within a class
2  $this  // in class C -> class(C)
3  self   // in class C -> class(C)
4  parent // in class C -> parentOf(class(C))
5  static // in class C -> class(C) or parentOf(class(C))
```

Listing 4.19: Special keywords

Class property fetch

* Possible add fact that the field E is declared in class C, when it is on the left side of an assignment.

$$\frac{\$this \to E_1 \subseteq C_1}{[E_1] = C_1.\text{hasProperty}(E_1.\text{name, static} \notin \text{Mfs}) \vee}$$
$$[E_1] = C_1.\text{parent.hasProperty}(E_1.\text{name, public|protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee$$
$$[E_1] = C_1[.\text{parent}].\text{hasMethod}("\_\_get")$$

$$\frac{self :: E_1 \subseteq C_1}{[E_1] = C_1.\text{hasProperty}(E_1.\text{name, static} \in \text{Mfs})}$$

$$\frac{parent :: E_1 \subseteq m}{[E_1] = C.\text{parent.hasProperty}(E_1.\text{name, static} \in \text{Mfs})}$$

$$\frac{E_1 \rightarrow E_2 \subseteq C_1*}{[E_1] = C_1.\text{hasProperty}(E_2.\text{name, static} \notin \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasProperty}(E_2.\text{name, public|protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee$$

$$[E_1] = C.\text{hasProperty}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})$$

*The same goes for static property fetches, except for the 'static $\notin$ Mfs' part: 'static $\in$ Mfs'.

$$\frac{E_1 \rightarrow E_2 \not\subseteq C \subseteq \Gamma*}{[E_1] = C.\text{hasProperty}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})}$$

*Property fetch outside a class scope, also for static properties.

```
1 $this->prop // name = prop, vis = public|protected , !static || mm
2 self::$prop // static property in class
3 parent::$prop // static property in the parent(s)
4 $a->prop // non-static property fetch
5 $a::$pro // static property fetch
```

Listing 4.20: Class property fetch

Class property fetch variable

$$\frac{E \equiv E_1 \rightarrow E_2, E_2 \text{ is an expression}}{[E_1] <: \text{object}()}$$

```
1 $b = "b";
2 $a->$b
```

Listing 4.21: Class property fetch variable

Class method call

$$\frac{E \equiv \$this \rightarrow E_1 \subseteq C_1}{[\$this] <: \text{object}(), [\$this] = \text{class}(C) \vee [E] :> \text{class}(C),}$$

$$[E_1] \text{ isMethod}(), [E_1] \text{ hasName}(E_1.name \vee "\_\_call"),$$

$$[E_1] = C_1.\text{hasMethod}(E_1.\text{name, static} \notin \text{Mfs}) \vee$$

$$[E] <: [E_1]$$

$$\frac{E \equiv self :: E_1 \subseteq C_1}{[E_1] = C_1.\text{hasMethod}(E_1.\text{name, static} \in \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name, public|protected} \in \text{Mfs} \wedge \text{static} \in \text{Mfs}) \vee$$

$$[E_1] = C_1.\text{hasMethod}("\_\_callStatic")$$

$$\frac{E \equiv parent :: E_1 \subseteq C_1}{[E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name, public|protected} \in \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasMethod}("\_\_callStatic")$$

$$\frac{E \equiv E_1 \rightarrow E_2 \subseteq C_1*}{[E_1] = C_1.\text{hasMethod}(E_2.\text{name, static} \notin \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasMethod}(E_2.\text{name, public|protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee$$

$$[E_1] = C.\text{hasMethod}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})$$

*The same goes for static method calls, except for the 'static $\notin$ Mfs' part: 'static $\in$ Mfs'.

$$\frac{E \equiv E_1 \rightarrow E_2 \not\subseteq C \subseteq \Gamma*}{[E_1] = C.\text{hasMethod}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})}$$

*method call outside a class scope, also for static methods.

This stuff is old!!!!!!

$$\frac{\$this \rightarrow E_1 \subseteq C_1}{[E_1] = C_1.\text{hasMethod}(E_1.\text{name}, \text{static} \notin \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name}, \text{public}|\text{protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee$$

$$[E_1] = C_1[.\text{parent}].\text{hasMethod}("\_\_\text{call}")$$

$$\frac{self :: E_1 \subseteq C_1}{[E_1] = C_1.\text{hasMethod}(E_1.\text{name}, \text{static} \in \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name}, \text{public}|\text{protected} \in \text{Mfs} \wedge \text{static} \in \text{Mfs}) \vee$$

$$[E_1] = C_1.\text{hasMethod}("\_\_\text{callStatic}")$$

$$\frac{parent :: E_1 \subseteq C_1}{[E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name}, \text{public}|\text{protected} \in \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasMethod}("\_\_\text{callStatic}")$$

$$\frac{E_1 \rightarrow E_2 \subseteq C_1^*}{[E_1] = C_1.\text{hasMethod}(E_2.\text{name}, \text{static} \notin \text{Mfs}) \vee}$$

$$[E_1] = C_1.\text{parent.hasMethod}(E_2.\text{name}, \text{public}|\text{protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee$$

$$[E_1] = C.\text{hasMethod}(E_2.\text{name}, \text{public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})$$

*The same goes for static method calls, except for the 'static $\notin$ Mfs' part: 'static $\in$ Mfs'.

$$\frac{E_1 \rightarrow E_2 \not\subseteq C \subseteq \Gamma^*}{[E_1] = C.\text{hasMethod}(E_2.\text{name}, \text{public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})}$$

*method call outside a class scope, also for static methods.

```
1 $this->methodCall();
2 self::methodCall();
3 parent::methodCall();
4 $a->methodCall();
5 $a::methodCall();
```

Listing 4.22: Class method call

Class method call variable

$$\frac{E \equiv E_1 \rightarrow E_2(), E_2 \text{ is an expression}}{[E_1] <: \text{object}()}$$

```
1 $a->$methodCall()
```

Listing 4.23: Class method call variable

Class constants (needs to be reviewed)

$$\frac{\text{self::}c_1 \subseteq \Gamma}{[\text{self::}c_1] = C_1.\text{hasConstant}(E_2.\text{name}) \vee}$$

$$[\text{self::}c_1] = C_1.\text{parent.hasConstant}(E_2.\text{name}, \text{public}|\text{protected} \in \text{Mfs})$$

$$\frac{\text{parent::}c_1 \subseteq \Gamma}{[\text{self::}c_1] = C_1.\text{parent.hasConstant}(E_2.\text{name}, \text{public}|\text{protected} \in \text{Mfs})}$$

$$\frac{E_1::c_1 \subseteq \Gamma}{[E_1] = \text{object}()}$$

```
1 self::CONST
2 parent::CONST
3 SOMECLASS::CONST
```

Listing 4.24: Class constants (needs to be reviewed)

Parameters

Parameters in class instantiation
*These parameters are just examples for what happens if they have typeHints (th), default values (v) or none *The constructor can be found in the m3 model (@constructors(loc classDecl, loc constructorMethodDecl))

$$\frac{\begin{array}{ll} new\ C_1\ (A_1,\ A_2,\ \ldots,\ A_k) \subseteq \Gamma & class\ C\ (th_1\ P_1,\ P_2 = v,\ \ldots,\ P_k) \subseteq \Gamma \\ \$a \to m()\ (A_1,\ A_2,\ \ldots,\ A_k) \subseteq \Gamma & public\ function\ m()\ (th_1\ P_1,\ P_2 = v,\ \ldots,\ P_k) \subseteq \Gamma \\ function_1\ (A_1,\ A_2,\ \ldots,\ A_k) \subseteq \Gamma & function\ (th_1\ P_1,\ P_2 = v,\ \ldots,\ P_k) \subseteq \Gamma \end{array}}{[P_1] <: [A_1],\ [A_1] <: [th_1],\ [P_1] <: [th_1],\ \text{hasRequiredParam}(P_1),\ \text{hasRequiredParam}(P_k)}$$

```
1 new C($foo);
```

Listing 4.25: Parameters in class instantiation

Scope

Type of a certain variable within some scope
this applies to global- class- function- and method- scope

$$\frac{E, E', E'', E''' \ldots etc \subseteq f \qquad E \text{ is a variable}}{[E] = [E] \vee [E'] \vee [E''] \vee [E'''] \ldots etc}$$

```
1 function f() {
2   $a = 1;
3   $a = "true";
4 }
5 // typeOf($a) is typeOf($a1, $a2, ..., $an);
```

Listing 4.26: Type of a certain variable within some scope

Return type of function or method (1) having no return statements or return;

$$\frac{\text{return} \nsubseteq f \vee \text{return}; \subseteq f}{[f] = null()}$$

```
1 function f() {} // no return = null()
2 function f() { return; } // return; = null()
```

Listing 4.27: Return type of function or method (1)

Return type of function or method (2) every exit path ends with a return statement

$$\frac{(\text{return } E_1) \vee (\text{return } E_2) \vee \cdots \vee (\text{return } E_k) \subseteq f}{[f] <: [E_1] \vee [E_2] \vee \cdots \vee [E_k]}$$

```
1 function f() {
2   if (rand(0,1))
3     return $a;
4   else
5     return $b;
6 }
7 // returns typeOf($a) or typeOf($b)
```

Listing 4.28: Return type of function or method (2)

Return type of function or method (3) possible no return value

$$\frac{(\text{return } E_1) \vee (\text{return } E_2) \vee \cdots \vee (\text{return } E_k) \vee (\neg\ \text{return}) \subseteq f}{[f] <: [E_1] \vee [E_2] \vee \cdots \vee [E_k] \vee null()}$$

```
1  function f() {
2    if (rand(0,1))
3      return $a;
4    else if (rand(0,1))
5      return $b;
6  }
7  // returns typeOf($a) or typeOf($b) or null()
```

Listing 4.29: Return type of function or method (3)

Function calls

Function call

$$\frac{f() \subseteq \Gamma}{[f()] <: \text{return of } [f]}$$

```
1  function f() {}
2  f();
```

Listing 4.30: Function call

Function call variable

$$\frac{E \equiv E_1() \subseteq \Gamma}{[E] = any(),}$$

$$[E_1] <: object() \vee [E_1] = string(),$$

$$if([E_1] <: object()) => hasMethod("\_\_invoke")$$

```
1  function f() {}
2  $f = "f";
3  $f(); // unknown what function will be called
4  // [$f] <: object(with __invoke method) | [$f] = string()
```

Listing 4.31: Function call variable

How to resolve expressions:
- Find all expressions which are defined above and annotate them with @type.
- Annotate the rest of the expressions with @type = any(); (should only be for relevant expressions)

4.3   Constraint solving

When we have all the constraints from the source code as facts, we will solve the constraints until we can no longer solve any constraints. The result will be a list of possible types for each class, method, fields, functions, variable and expression. The first step is to initialise all type-able objects we want to solve. In this initial phase the status of all objects can be of any type, because we do not know anything about them yet. When we solve the constraints step by step, we will be able to limit the number of possible types for a certain object. We do this by taking the intersection of the constraint result and the possible types we have. This way there should be less and less possible types for each variable.
When we take the intersection of none overlapping types, we have a type error. There is no possible type for this object. Because PHP allows ducktyping, allowing objects to be of more than one type, we will use widening in these cases. Widening means that we pick the union instead of the intersection, so we can continue our analysis.
Some of the constraints only give us information that a certain object should be a subtype of a certain type. When we solve these constraints, we will take the least common ancestor.

4.4   Annotations

After we gathers all the facts from the source code, we will add additional information which we read from the annotations. For this we use regex to match @return type and @param type var for methods and functions. We read @var type for variables and class attributes. In our first analysis we do not include the facts we gathered from the annotations. In the second analysis we do include the facts. This way we can compare the end results.
In order to gain some knowledge about the reliability of the annotations, we compare the result our or initial analysis with the provided provided annotation information. Here the implementation should comply to the used annotations.

Chapter 5

Analysis

In order to validate the performed research we have tested them on the most popular packages of Packagist[1], which are listed in table 5.1. The statistics are generated using phploc[2]. All packages have between 2 and 6 million downloads. Packagist is a repository for Composer[3]projects. Composer is a dependency manager for PHP projects. All external plugins for PHP projects can be managed via a composer.json file. You only need know the name and a version of the external package in the repository of your project.

Product			Files		Objects			Lines of code				
Vendor	Project*	Version	D[1]	F[2]	C[3]	I[4]	T[5]	Total ↑	Logical		Global[6]	
doctrine	lexer	v1.0	2	7	3	0	0	733	128	(17.46%)	13	(10.16%)
phpunit	php-timer	1.0.5	5	11	5	0	0	740	117	(15.81%)	17	(14.53%)
phpunit	php-text-template	1.2.2	5	11	5	0	0	768	125	(16.28%)	15	(12.00%)
doctrine	inflector	v1.0	2	7	3	0	0	853	130	(15.24%)	13	(10.00%)
psr-fig	log	1.0.0	3	15	8	2	2	1 039	155	(14.92%)	22	(14.19%)
phpunit	php-file-iterator	1.3.4	5	13	7	0	0	1 071	176	(16.43%)	15	(8.52%)
symfony	filesystem	v2.5.3	3	11	5	2	0	1 090	193	(17.71%)	19	(9.84%)
symfony	yaml	v2.5.3	3	16	11	1	0	2 270	509	(22.42%)	28	(5.50%)
phpunit	php-token-stream	1.2.2	6	13	169	0	0	2 360	377	(15.97%)	15	(3.98%)
doctrine	collections	v1.2	3	18	11	3	0	2 504	394	(15.73%)	33	(8.38%)
symfony	process	v2.5.3	3	19	14	1	0	3 198	604	(18.89%)	37	(6.13%)
symfony	finder	v2.5.3	8	43	36	3	0	4 976	909	(18.27%)	80	(8.80%)
symfony	dom-crawler	v2.5.3	12	63	53	6	0	7 825	1 296	(16.56%)	157	(12.11%)
symfony	translation	v2.5.3	21	121	97	20	0	12 345	2 299	(18.62%)	257	(11.18%)
symfony	console	v2.5.3	17	84	66	13	2	13 546	2 556	(18.87%)	246	(9.62%)
symfony	http-foundation	v2.5.3	16	90	76	10	0	14 179	2 262	(15.95%)	154	(6.81%)
twig	twig	v1.16.0	18	172	148	19	0	14 689	2 630	(17.90%)	15	(0.57%)
symfony	event-dispatcher	v2.5.3	27	170	133	31	3	20 230	3 629	(17.94%)	418	(11.52%)
swiftmailer	swiftmailer	v5.2.1	37	238	170	52	0	28 965	4 645	(16.04%)	144	(3.10%)
phpunit	php-code-coverage	2.0.1	62	259	381	24	0	50 371	6 579	(13.06%)	87	(1.32%)
phpunit	phpunit	4.2.2	65	270	388	26	0	51 516	6 764	(13.13%)	129	(1.91%)
phpunit	phpunit-mock-objects	2.2.0	66	271	393	27	0	51 735	6 801	(13.15%)	132	(1.94%)
doctrine	annotations	v1.2.0	69	306	423	28	0	57 325	7 718	(13.46%)	188	(2.44%)
doctrine	common	v2.4.2	76	337	440	45	0	62 406	8 326	(13.34%)	298	(3.58%)
symfony	http-kernel	v2.5.3	96	565	471	90	3	79 294	14 169	(17.87%)	1 449	(10.23%)
doctrine	cache	1.3.0	152	687	729	102	2	103 024	16 667	(16.18%)	1 355	(8.13%)
doctrine	dbal	v2.4.2	121	557	628	63	0	104 630	15 234	(14.56%)	1 033	(6.78%)
guzzle	guzzle	v3.9.2	150	832	828	141	7	117 699	19 772	(16.80%)	1 787	(9.04%)
doctrine	orm	v2.4.4	175	1007	875	119	2	158 530	27 932	(17.62%)	2 866	(10.26%)
monolog	monolog	1.10.0	350	1911	1 904	135	2	288 507	31 415	(10.89%)	4 221	(13.44%)
werkspot	old-Website	07-2014	928	6225	4 907	224	0	1 054 686	167 978	(15.93%)	22 693	(13.51%)

*This is a list of the 30 most popular packages of packagist ordered by total lines of code, in July 2014.
[1] = Directories, [2] = Files, [3] = Classes, [4] = Interfaces, [5] = Traits, [6] = Not in class or function

Table 5.1: List of analysed projects.

To collect the source code for each project, we have executed the following steps:

1. `git clone` the github repo.

2. Run `composer install`, and the source code including dependencies will be downloaded in the `/vendor` folder.
3. Remove the `autoload.php` and `composer` folder, as we don't need them.
4. Remove the test folders by removing all folders matching `Tests` or `tests`.

To measure the coverage:

1. `git clone` the github repo.
2. Run `composer install`, and the source code including dependencies will be downloaded in the `/vendor` folder.
3. Run the unittests and use xdebug to resolve the types.
4. Compare the results.

Chapter 6

Results

For the results we picked X software products to see how it performs. For each product we performed the type inference with and without reading annotations from the doc blocks.

6.1 Results

Show the results...

6.2 Validation of the results (or something)

Say something about:

- Soundness (what we measured, is it correct?)
- Completeness (how much did we measure?)
- Accuracy (how precise are the results?)

6.3 Annotations

The results of the analysis when adding the annotations to the analysis. Compare the results with the results of the analysis without the annotation information.

Chapter 7

Case Study

Explain how the case study is performed.

This chapter will show the case study, but I just need to place this information somewhere.

A list of the 40 most popular packages from packages.

- create composer file

- composer install

- mkdir phploc

- list all packages: find ./vendor/* -maxdepth 1 -mindepth 1 -type d -exec ls -d ""

- prefix the list with "phploc" and postfix with " > phploc/<file>.phploc"

Chapter 8

Conclusion

Summary of the whole work, with conclusions. T.B.A.

8.1 Conclusion

8.2 Future work

These items will not be covered by the analysis (maybe add this to threats/future work)

- Analysis is flow insensitive

- Closure

- References

- Variable constructs (variable -variable, -method/function calls, -class instantiation, eval) :: todo: explain WHY not.

- Yields

Explain something about combining this analysis to other analysis (like dead code elimination, constant folding/propagation resolve, alias analysis, array analysis) to gain more precise results.

Something about performance optimisations... Explain what is already done to boost the performance and what still can be done.

Use a bigger corpus to gains better results of the analysis by doing analysis on more programs.

8.3 Threats to validity

When relying on annotations for analysis, the annotations need to be correct and sound. The hard part is that the annotations are not examined on runtime, and are therefor not easily validated.

Glossary

AST

An abstract representation of the structure of the source code. .

Rascal

Rascal is a meta-programming language developed by SWAT (Software analyse and transformation) team at CWI in the Netherlands. See http://www.rascal-mpl.org/ for more information.

reflexive transitive closure

A relation is transitive if $\langle a, b \rangle \in R$ then $\langle b, a \rangle \in R$.
A relation is reflexive if $\langle a, b \rangle \in R$ and $\langle b, c \rangle \in R$ then $\langle a, c \rangle \in R$.
A reflexive transitive closure can be established by creating direct paths for all indirect paths and adding self references, until a fixed point is reached.

stdClass

A predefined class in the PHP library. The class is the root of the class hierarchy. It is comparable to the Object class in Java.

Bibliography

[Big10]   Paul Biggar. "Design and Implementation of an Ahead-of-Time Compiler for PHP". In: (2010).

[HKV13]   Mark Hills, Paul Klint, and Jurgen J. Vinju. "An Empirical Study of PHP feature usage: a static analysis perspective". In: *ISSTA*. Ed. by Mauro Pezzè and Mark Harman. ACM, 2013, pp. 325–335.

[Izm+13]   Anastasia Izmaylova et al. "M3: An Open Model for Measuring Code Artifacts". In: *CoRR* abs/1312.1188 (2013).

[KSK10a]   Etienne Kneuss, Philippe Suter, and Viktor Kuncak. "Phantm: PHP Analyzer for Type Mismatch". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 373–374. ISBN: 978-1-60558-791-2. DOI: 10.1145/1882291.1882355. URL: http://doi.acm.org/10.1145/1882291.1882355.

[KSK10b]   Etienne Kneuss, Philippe Suter, and Viktor Kuncak. "Runtime Instrumentation for Precise Flow-Sensitive Type Analysis". English. In: *Runtime Verification*. Ed. by Howard Barringer et al. Vol. 6418. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 300–314. ISBN: 978-3-642-16611-2. DOI: 10.1007/978-3-642-16612-9_23. URL: http://dx.doi.org/10.1007/978-3-642-16612-9_23.

[KSV09]   Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *SCAM*. 2009, pp. 168–177.

[Zha+12]   Haiping Zhao et al. "The HipHop Compiler for PHP". In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 575–586. ISSN: 0362-1340. DOI: 10.1145/2398857.2384658. URL: http://doi.acm.org/10.1145/2398857.2384658.