Type inference for PHP
A constraint based type inference written in Rascal

Ruud van der Weijde

September 30, 2016, 48 pages

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

Werkspot

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

Despite the wide usage of PHP programs over the internet, there is a lack of good analysis tools. Dynamic language like PHP are generally hard to statically analyse because of run-time dependencies. Without running the program there are many things undecided. In this thesis we present a constraint based type inference written in Rascal. Rascal is a programming language for meta-programming in the domain of software analysis and transformations. We show that the number of inferred types increases when type hint annotations are taken into account.

Preface

This thesis could not have been completed without the help and support of various people. In this section I want to thank people who helped me during the process by mentioning their name and describe how they helped me. I start with a short story on how this research came to be.

The initial idea for this thesis was to dive into the topic of software analysis in order to find vulnerability in PHP programs. When diving further in this topic my interest in statical analysis grew, and I wanted to reconstruct data flows in a program to be able to perform taint analysis. In order to perform a dataflow analysis for object oriented programs written in PHP, the types of variables need to be known. This topic of resolving the run-time type at compile time was big enough to have a thesis on its own.

First of all I would like to thank Jurgen Vinju whoem helped me throughout the whole thesis process. I want to thank Jurgen for his endless enthusiasm, number of ideas and suggestions and personal help on coaching and mentoring me. Jurgen helped defining a research topic and gave me many directions where to go or look for whenever I got stuck on a tough subject.

Next I would like to thank Mark Hills. The implementation of this research is build on top of the PHP Analysis in Rascal (PHP AiR) framework, created by Mark. We used the framework to parse PHP files to Rascal ASTs and implement an M^3 model for PHP programs.

I want to thank Bas Basten for the collaboration on improving and extending the M^3 model. After I made an initial version of the M^3 model for PHP, Bas help to improved the model by adding more facts to the model. With his expertise in Rascal and and my knowledge of PHP we've formed a very good team.

I would like to thank Winfred Peereboom for giving me the opportunity to write my thesis at Werkspot. During the period Winfred helped me on various aspects of coaching and mentoring me.

Finally I would like to thank my girlfriend for her endless support and patience. And of course I want to thank everyone I forgot to mention here personally, but did help me direct or indirect, consciousness or subconsciousness.

Chapter 1

Introduction

1.1 PHP

PHP[1]is a server-side programming 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 and allows objects to be changed during run-time. PHP uses duck-typing which means that there is no type checking on objects type, but only checks if the attempted operation is permitted on the object.

Evolution The programming language PHP evolved since its creation in 1995. The first milestone was in the year 2000 when Object-Oriented (OO) language structures were added to the language with the release of PHP 4.0. The 5th version of PHP, released in 2004, provided an improved OO structure including the first type declarations for function parameters. Namespaces were added in PHP 5.3 in 2009, to resolve class naming conflicts between libraries and to create better readable class names. The 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 latest 5.x version is 5.6 and includes more internal performance optimisations and introduces a new debugger. The most recent stable version is 7, which is not taken into this research. In this lastest version they've achieved a mayor performance increase and memory decrease. Also type hints for scalar types are added, method/function return types, and strict typing can be enforced for the available type hints.

Popularity According to the Tiobe Index[2]of July 2016, 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 running php backend[3]. The majority of these websites use PHP version 5, rather than older or newer versions. We therefor focus our type inference on PHP version 5.

1.2 Position

Although the popularity of PHP for about 15 years, there is still a lack of PHP code analysis tools with full language support. Due to various dynamic features in PHP not all types and execution paths can be resolved without actually executing the program. Source code analysis tools need to know execution paths and types or expressions for optimal results in discovering security vulnerabilities or bugs. Such tools could also provide code completions or do automatic transformations when executing refactoring patterns. Source code analysis is performed statically or dynamically or a combination of the two. In static analysis the program is not executed.

[1]http://php.net
[2]http://www.tiobe.com/tiobe_index, July 2016
[3]http://w3techs.com/technologies/details/pl-php/all/all, July 2016

1.3 Contribution

This research contributions to the static analysis research field of dynamic programming languages by presenting a type inference analysis where we over-approximate runtime values at compile time. Results of this analysis can be used to improve software analysis tools. In order to resolve the types of expressions we've implemented a generic model, M^3, that holds various facts about programs with support for PHP programs.

The main contributions of this thesis are:

- M^3 model for PHP programs
- constraint based type inference for PHP programs

An M^3 *model for PHP programs* contains various facts about the programs. The M^3 model, see section 5.1 for more information, was initially solely supporting Java programs. The addition of support for PHP programs was beneficial not only for this thesis, but also for other researchers. The model provides program context while extracting and resolving constraints in this research. The model helped other researchers to compare the usage of PHP programs with programs written in other languages.

The biggest contribution of this thesis is the *constrained based type inference for PHP programs*. In this type inference process we use an abstract syntax tree to generate type constraints on language constructs. We then solve these constraints to come to a set of types for each expression. We variated the inference process by adding context information of type annotations and php built-in to the constraint extraction to find out that this helps to resolve more types. The inferred types can help IDE tools and programmers by providing helpful tools, which could lead to better software development.

1.4 Plan

The rest of this thesis is as follows:

Chapter 2 contains background and related work. Background consists of important language constructs, information about annotations in PHP, introduction to Rascal and M^3 and type systems. We end this chapter with related researches and their relation to this research.

Chapter 3, research context, describes the research approach and context. We explain under which assumptions this research is executed.

Chapter 4 describes the design of the type inference for PHP. We present the constraint rules on various language constructs. Next to that we give information about type annotations and php built-in information.

Chapter 5 contains implementation details. We show how we implemented the M^3 model for PHP. Next we explain how we implemented the constraint extracting. The constraint solving is explained by showing the used algorithm.

Chapter 6 shows type inference results on real world PHP programs. We present the results of multiple programs and analyse the results.

Chapter 7 ends this thesis with the conclusion, future work section and lists various threats to validity.

Chapter 2

Background and Related Work

Chapter 2 provides relevant background information. The first section, section 2.1, describes seven important language constructs which the reader needs to understand in order to understand the difficulties of analysing PHP. Section 2.2 explains Rascal, the programming language used for the analysis. In this section we will explain M^3, a programming language independent meta model which holds various facts about programs, in more details. Section 2.3 provides information about type systems and how type systems relate to this research. The last section of this chapter, section 2.4, presents the related work and how these researches relate to this thesis.

2.1 PHP Language Constructs

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

Scope 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 upcoming paragraph about conditional classes and functions) will be available in the global scope. If a class or function is declared inside a namespace, their full qualified name includes 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].

Script includes PHP allows files to include other PHP-files during program execution. The content of these files will be loaded at the place where the include statement is defined. This means that if you use an include in the middle of a file, the source code of this included file will be virtually inserted at that position.
File includes are mainly used for loading classes and for including templates to render output. PHP5 allows automated class loading based on the namespace, which is called autoloading classes. With autoloading classes there is no need for including files manually for each class.
Research by Mark Hills et al.[HKV14] has shown that most includes can be resolved with statical analysis. In this research we do not run such an analysis, we will assume that all files in the project are included during execution.

[1] http://php.net/manual/en/language.variables.scope.php, July 2016

Conditional classes and functions Once a file is included in the execution, al the found classes and functions are declared in the top level scope. All class and function declarations within condition statements or within a method or function scope are only declared when the conditional statement is executed.
An example of an conditional statement can be found in listing 2.1. If the class `Foo` or function `bar` do not exist before the statements are executed, then the class and function will not yet be declared. When you try to use the class or function before the code is executed, the script will exit with an fatal error.

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

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

Listing 2.1: Conditional class and function definitions

Another example of dynamic function and class loading is displayed in listing 2.2. If the first call is `g()` as you can see in `line 8`, the script will result in a fatal error because `function g()` will only be declared after `function f()` is executed. `class C` will be declared once `function g()` is executed. As soon as the functions and classes are declared, they are available in the top scope, possibly prefixed with the name of the namespace they are declared within.

```
1  function f() {
2    function g() {
3      class C {}
4    }
5  }
6
7  // Execution examples:
8  g();                  // will fail because 'g();' is not declared yet
9  f(); g();             // will work because 'g();' is declared when calling 'f();'
10 f(); new C();         // will fail because 'g();' needs to be called first
11 f(); g(); new C();    // will work because 'g();' is called and has declared 'f();'
```

Listing 2.2: Conditional function declaration

Dynamic features PHP comes with dynamic built-in features like: include dynamic variables, dynamic class instantiations, dynamic function calls, dynamic function creation, reflection, and eval. New functions and classes can be declared on the fly during run-time. Method calls, or even whole pieces of code, can be executed based on variable strings.
A previous study by Mark Hills[HKV13] has shown that most real applications make use of dynamic features. Dynamic features are powerful, but can complicate the statical analysis. Analysis like constant propagation is needed to help resolving most of these dynamic features. This is not in scope for this research, but could be added in future work.

Late static binding Late static binding[2]is implemented in PHP since version 5.3 by adding the keyword `static` to the language. It's usage is similar to the keyword `self`, which refers to the current class. The main difference is that `self` refers to the class where the code is located, while `static` refers to the actual instantiated class. The keyword `self` can be statically resolved while `static` can only be resolved on runtime.

Magic methods PHP allows calls and property access on methods and fields that don't exist on a class. Normally a call to a non-existing method or property would result in a fatal error, but with the use of magic methods you can specify the wanted behavior. Listing 2.3 shows an example of the `__call` method. This method is triggered when a non-accessible or non-existing method is called. In this example the code will try to return the value of a private property based on the provided name. The

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

full list of magic methods is `__construct`, `__destruct`, `__call`, `__callStatic`, `__get`, `__set`, `__isset`, `__unset`, `__wakeup`, `__toString`, `__invoke`, `__set_state`, `__clone`, and `__debugInfo`.

```
class Car {
    private $maxSpeed = 210;
    function __get($name) { return $this->$name; }
}
var_dump((new Car)->maxSpeed); // 210
var_dump((new Car)->numberOfWheels); // NULL
```

Listing 2.3: Magic methods in PHP

Dynamic class properties Although it is a good practice to define your class properties, it is not required to do so in PHP. After instantiating a class it is possible to add properties the objects, even without the implementation of magic methods. In listing 2.4 you can see a code sample of adding a property to an object. The access of the non-existing property `nonExistingProperty` will result in a warning, but code execution will continue and will just return `NULL`. The code on `line 4` 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.

```
1 class C {}
2 $c = new C();
3 var_dump($c->nonExistingProperty); // NULL
4 $c->nonExistingProperty = "property now exists";
5 var_dump($c->nonExistingProperty); // string(19) "property now exists"
6 var_dump((new C)->nonExistingProperty); // NULL
```

Listing 2.4: Dynamic class property

2.2 Rascal

Rascal[KSV09] is a meta programming language developed by Centrum Wiskunde & Informatica (CWI). Rascal is designed to analyse, transform and visualise source code. The language is build on top of Java and implements various concepts of existing programming languages. In this research, Rascal is the main programming language. Rascal is used for gathering facts about the program and to solve constraints. The facts are gathered by visiting AST tree representing the program and hold semantic information about the program. Constraints are generated based on the collected facts and these constraints are solved with an in Rascal created constraint solver. The only part that does not use Rascal is the PHP parser. Although this could be easily implemented in Rascal, there was an existing library written in PHP available.

***M*³**[Izm+13; Bas+15] is a model which holds various information of source code and is implemented in Rascal. This model is created to gain insights in the quality of open-source projects. For our research we use the M^3-model to store facts about the program in a structured way, so we can easily use it at a later stage.

The core elements of the M^3-model are `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 and its source location. 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 references use an 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, and info statements.

2.3 Type systems

A **Type** is a set of possible values and a set of operations that can be performed on them. PHP is a dynamically typed language, which means that the types of the expressions are not examined at compile time. PHP implements duck typing, so the type of objects are not examined during run-time, but only checks if the operations are allowed on the objects. In this thesis we are interested in the run-time types of values so we are able to perform statical source code analysis.

Type systems Type systems define how a set of rules are applied to types in their context. A type system validates the type usage with **type checking**. In order to perform type checking the types of the expressions needs to be known. The process of resolving the types of the expression is called **type inference**. Both type checking and type inference are explained in more details below.

Type checking Type checking is a mechanism which validates and/or enforces the constraints of a type in their specific context. There is a difference between static type checking and dynamic type checking. **Static type checking** is a process of checking the types based on the source code. The static type checker will ensure that a program is type safe before executing the program, which means that there will occur no type errors during runtime. **Dynamic type checking** performs the type checking during runtime. This means that the program needs to run to gain feedback on the usage of types. PHP is a dynamically typed language, which means that there are no types checked before actually running the program. Although PHP also implements some static type features, like parameter type hints, it cannot be perfectly determine all types at compile time.

Type inference Type inference is the process of resolving types of variables and expressions. The inference process is a prerequisite to perform type checking. Being able to infer the type before running the program enables you to optimise code execution by applying compiler optimisations. These optimisation allow performance improvements or can optimise memory usage. In dynamic languages like PHP it can be difficult to resolve the type of a variable or expression without running the program. In statically typed languages, type inference happens at compile time. In the next paragraph we will briefly explain some type inference systems.

The **Hindley-Milner**[Hin69] (HM) type system was found in 1969 by Roger J. Hinldey and almost 10 years later rediscovered[Mil78] by Robin Milner. The first implementation was created four years later by pHd student Luis Damas. Damas proved the soundness and completeness of the HM type system with **Algorithm W**[DM82] in the context of the programming language `ML`. The HM type system deduces the types of the variables to their most abstract type, based on their usage. Type declarations and hints are not necessarily to perform type inference. The type system is used for various functional languages. `Haskell` for example uses the Hinley-Milner type system as a foundation for the `Haskell` type system.

Control Flow Analysis[NNH99] (CFA) is concerned with resolving sound approximate run-time values at compile time. CFA is build on top of data flow analysis[ASU86] and tries to resolve the control-flow problem for high order programming languages. The control-flow problem deals with resolving which caller can call which callee in a program. One of the earlier CFA algorithms was Shivers' 0CFA algorithm[Shi88], a flow-sensitive constraint based algorithm. Shiver then defined k-CFA[Shi91], where the precision of the analysis is increased by taking the context of the expressions into account. The k-CFA algorithms compute an conservative over-approximations of run-time values during compile type.

The **Cartesian Product Algorithm**[Age95] (CPA) is a type inference algorithm created by Ole Agesen in 1995. Agesens work was based on Palsberg and Schwartzbach' **basic type inference algorithm**[PS91]. This basic type inference algorithm derives a set of constraints based on *trace graphs* and solves the constraints using a fix-point algorithm. Agesen extended the basic algorithm with *templates*. These templates are based on control flow and have start and end nodes with their possible in- and outputs. The CPA calculates the possibles output types for each template by taking the cartesian product, the set of all possible ordered pairs, of the input types.

2.4 Related work

In this section we will briefly describe related research work in order to get a better understanding similar performed researches in the same research area.

Similar work has been presented by Patrick Camphuijsen[Cam07; CHH09]. Patrick created a constraint-based type inference analysis for his master's thesis. The inference algorithm combines possible results of the constraints and takes the union to define the types. To guarantee termination the algorithm uses widening, by replacing the current result with the result of the union, to make sure that there will be a fixed-point. Further work improved the implementation by adding support for objects[VH15].

Paul Biggar created an Ahead-Of-Time (AOT) compiler for PHP[Big10]. The main goal of this compiler is to improve the performance of PHP programs. The AOT compiler starts by parsing a PHP program into an AST. This AST is transformed into an High-level Intermediate Representation (HIR) to remove all redundant constructs and then transformed into a Medium-level Intermediate Representation (MIR). Using dataflow analysis, alias analysis, static single assignment (SSA), and type analysis the compiler performs optimisations on the MIR. After the optimisations, the compiler generates C code, which then can be executed to run the program.

`PHANTM`[KSK10a; KSK10b] (PHp ANalyzer for Type Mismatch) is an open source PHP analyser written in Scala. Because of PHP's dynamic nature, without compiler or interpreter type checking, it is easy to make typing errors that result in unexpected behaviour or in fatal errors. `PHANTM` performs a hybrid flow-sensitive analysis to find type errors in PHP5. The hybrid analysis combines static and dynamic analysis. A program can be annotated to start a static analysis at a specific point. The analyser collects run-time type information while running the program and then starts the static analysis. `PHANTM` uses data-flow analysis to infer types. Although `PHANTM` has proven to be able to find a decent number of type errors on scalar usages in three different programs, there is a lack of finding errors in object oriented structures.

Facebook improved the performance of PHP programs with a static compiler, called HipHop Virtual Machine[Zha+12] (HHVM). This static compiler extracts the program into an AST, traverses this AST to collect information, performs pre-optimisations, performs type inference, preforms post-optimisations, and lastly generates C++ code. During the pre-optimisations the compiler removes unneeded actions, for example constant inlining, logical-expression simplifications, and dead-code elimination. The type inference process is based on the Hindley-Milner constraint based algorithm[DM82], to infer types of constants, variables, functions parameters, and return types. These new inferred types are then used in the post-optimisation. In the last step the AST is traversed to generate C++ code. Although the compiler does not cover all functions of PHP, it does covers most of the features. The performance benefits on the other hand are significantly better, showing on average 5.5x more efficiency for PHP5.

PHPLint[3]extends the PHP syntax with type hints where PHP lacks support for it, using custom inline comment blocks to add extra typing information. These doc blocks with type information can be used in the analysis, allowing more strict type checking. The used syntax for the type hints are `/*. .*/`, for example: `/*. string .*/ $s = $var;` which means that the variable `$s` is of type `string`. PHPLint solves the lack of type hint support on scalar and array types. PHPLint can generate type hints based on information retrieved from simple type inference.

John Aycock states in Aggressive Type Inference[Ayc00] that dynamically typed languages are also used as statically typed languages. This means that you can use dynamic languages without using the magical parts. This claim is supported by a small analysis on the usage of `eval`, `execfile`, and `import`. Due to this statement he advocated for type consistency within a scope for variables, because the usage of the magic methods is limited. His analysis is flow insensitive, which means that it ignored specific paths within a scope.

[3]http://www.icosaedro.it/phplint, July 2016

Chapter 3

Research Context

This chapter describes our type system of PHP the research context. In section 3.1 we will explain more about the types we have defined. The relation between the types are described in 3.2. Section 3.3 explains in which context the research is executed.

3.1 Types

The basis types in PHP are integers, floats, booleans, strings, arrays, resources and null. 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.
Because PHP has no explicit type system, we define our own type system for PHP. In the Rascal code below you can see our defined types, with a brief description below.

Rascal 1 M^3 core definitions in Rascal

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

data TypeSymbol
  = \any()[1]                          // unknown, can be any of the types below
  | arrayType(TypeSymbol arrayType)    // array of a type, can be nested
  | booleanType()                      // boolean values
  | classType(loc decl)                // a specific class
  | floatType()                        // float, double or real
  | integerType()                      // integer numbers
  | interfaceType(loc decl)            // a specific interface
  | numberType()                       // a float or integer
  | nullType()                         // empty or undefined value
  | objectType()                       // any class type
  | resourceType()                     // a build-in type
  | scalarType()                       // any number, string, resource or
  | stringType()                       // text values
  ;
```

any As you can see in the comments in the code above, Rascal 1, `any()` represents the combination of all possible types. This type will be used for mixed and unknown types, for example when variables are used, but are never defined.

arrayType The type `arrayType(TypeSymbol arrayType)` is a recursive declaration. The argument of the type is the type of the array. For example, an array of strings is declared as `arrayType(stringType())`

and for an unknown array the type is `arrayType(\any())`.

booleanType The type `booleanType()` is the type for boolean values. Just like any other language the boolean values are `true` and `false`.

classType The type `classType(loc decl)` represents a specific class, or the generic type. The argument is the declaration, which represents the logical name of the class. An example of the `Exception` class is `classType(|php+class:///exception|)`.

floatType Floating point numbers, also known as floats, reals, and doubles are defined by the `floatType()`. Example are `1.234`, `1.2e3`, and `7E-10`.

integerType Integers are whole numbers in decimal, hexadecimal, octal or binary notation. The `integerType` values can be positive or negative.

interfaceType The `interfaceType()` represents a specific interface. Interfaces can be provided as type hints.

numberType The type `numberType()` covers the `integerType` and `floatType`. Because of coercion, these types can be easily mixed.

nullType The type `nullType()` is used for the value `null`.

objectType The type `objectType()` is the parent type for all class types. This type represent the object type and could also been written as `classType(|php+class:///object|)`.

resourceType The type `resourceType()` represents the build-in PHP resource type. Various function return the resourceType from build-in PHP functions.

scalarType The type `scalarType()` is the generic type for `resourceType`, `booleanType`, `numberType`, and `stringType`.

stringType The type `stringType()` represents strings, a sequence of characters.

3.2 Type hierarchie

As already described in some of the previous descriptions, types can relate to other types. The relation between the types is shown in figure 3.1. In this diagram the `-Type` postfix is omitted to save space. We speak of **subtypes** when the types are descendant of the given type. The subtypes of the root node `any` are `scalarType`, `arrayType`, and `objectType`.

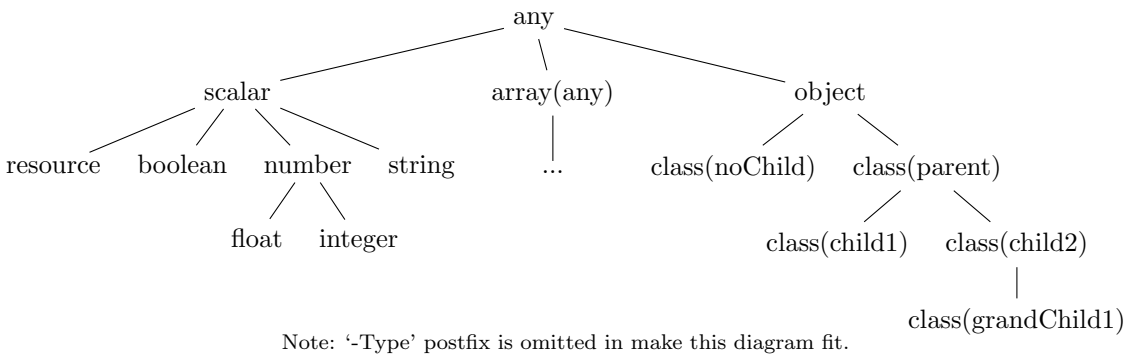
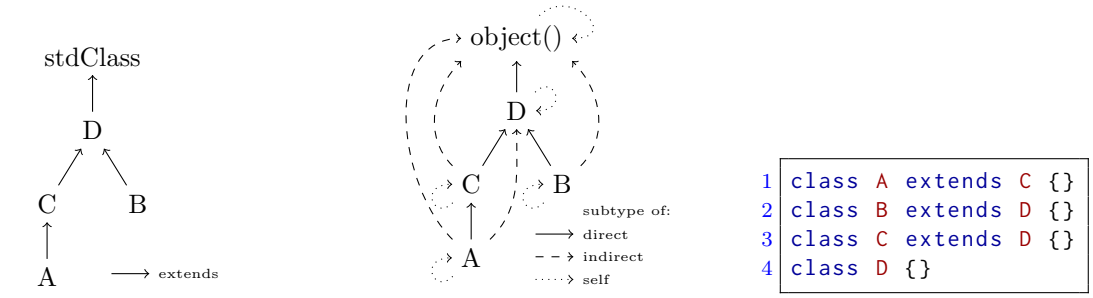


Figure 3.1: Type hierarchy

The *scalar type* is the super type for the non-complex types `resourceType`, `booleanType`, `numberTypes`, and `stringType`. These types can in practise be combined because of coercion. If they are used together, they will be classified as scalar types.

The *array type* in the subtype diagram is the most generic type of array, the array of any type. We have omitted the other array types to reduce complexity of the hierarchy. In theory, this array type is a recursive type and can go to infinite depth.



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

Figure 3.2: Relation of subtypes among classes

The *object type* is the most generic object type, which represents the `stdClass` in PHP. The class inheritance relation in PHP 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.

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 assume that the programs are correct and works as intended. This is needed to be able to reason about the programs we analyse, without having to question wether the program works as intended or not.

File includes In this research we will assume that all PHP file are included during runtime within the project folder. 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 our experiment we are sure that all chosen projects comply to this.

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. This feature is disabled by default in version 4.2 and has been completely removed in version 5.4.

PHP warnings For this research we will ignore all warnings. Warnings do not alter the behaviour of the program. In a most production environment these warnings are suppressed and will not change the behaviour of the program. We do take fatal errors into account, which lead to runtime errors.

Flow insensitive Our analysis is intra-procedural flow-insensitive, which means that we don't take the order of statements into account within a function or method scope. We do assume type consistency within a scope.

Chapter 4

Design of PHP type inference

In this chapter we present the type constraint rules, in section 4.1, for PHP language constructs with supporting code examples. In the last two sections we provide more information on annotations in section 4.2, and on PHP built-ins in sections 4.3.

4.1 Type inference rules

The constraint definitions we use in this section are based on the definition of Palsberg and Schwartzbach[PS94]. We have extended the definition to conform to the PHP language. A legend with all symbols is displayed in table 4.1, followed by the constraint definitions for PHP.

symbol		description	symbol		description
$\equiv$	=	equivalent expression	$=$	=	equivalent type
$:=$	=	assignment	C	=	a class
$<:$	=	(lhs) is subTypeOf (rhs)	$\rightarrow c$	=	a class constant
E_k	=	an expression	$\rightarrow p$	=	a class property
$[\![E_k]\!]$	=	typeset of an expression	$\rightarrow m$	=	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	$\in$	=	is defined in
$\{\}$	=	set of types	`is_a`	=	equal to PHP code
$\wedge$	=	conjunction	$\vee$	=	disjunction

Table 4.1: Constraint definition legend

We write the definitions in the following form:

$$\frac{\text{premiss 1} \qquad \text{premiss 2}}{\begin{array}{c}\text{constraint 1,}\\ \text{constraint 2}\end{array}}$$

Above the horizontal line we write the premisses. In our case premisses are PHP expressions which are true or false depending on the context. If the premiss is true for a PHP statement or expression we can define the constraints below the horizontal line.

4.1.1 Scalars

Extracting constraints from the scalar types is straight forward. We show the constraint rules for strings, integers, floats, booleans, and null values.

Strings Strings in PHP can be written with single or double quotes. If an expression is a literal string we can add the constraint that the typeset of that expression is equal to a string type.

$$\frac{E \texttt{ is\_a } \text{string}}{[\![E]\!] = \{ \text{ stringType() } \}}$$

Code sample:

```
"Str"; // stringType()
'abc'; // stringType()
```

Listing 4.1: Strings

Integers In the examples below you can see different types of integers. If we encounter a expression that represents one of these integer formats, we can extract the constraint that the typeset of the expression should be equal to an integer type.

$$\frac{E \texttt{ is\_a } \text{integer}}{[\![E]\!] = \{ \text{ integerType() } \}}$$

Code sample:

```
1234;       // integerType() (decimal number)
-123;       // integerType() (negative number)
0123;       // integerType() (octal number)
0x1A;       // integerType() (hexadecimal number)
0b11111111; // integerType() (binary number)
```

Listing 4.2: Integers

Floats If we see php syntax that represents a floating number, we can extract the constraint that this expression if of floating type.

$$\frac{E \texttt{ is\_a } \text{float}}{[\![E]\!] = \{ \text{ floatType() } \}}$$

Code sample:

```
1.4;   // floatType()
1.2e3; // floatType()
7E-10; // floatType()
```

Listing 4.3: Floats

Boolean values Boolean values in PHP are case sensitive, as you can see in the examples. If we encounter a boolean value, we can extract the constraint the this expression is of boolean type.

$$\frac{E \equiv (\text{true}|\text{false})}{[\![E]\!] = \{ \text{ booleanType() } \}}$$

Code sample:

```
true;  // booleanType()
false; // booleanType()
TRUE;  // booleanType()
FALSE; // booleanType()
```

Listing 4.4: Boolean values

Null values `null` is a reserved keyword in PHP. When we encounter null in the source code we can add the nullType type constraint.

$$\frac{E \equiv \text{null}}{[\![E]\!] = \{\ \text{nullType}()\ \}}$$

Code sample:

```
null;  // nullType()
NULL;  // nullType()
```

Listing 4.5: Null values

4.1.2 Assignments

Assign statements transfer values from one expression or variable into another. PHP uses the = symbol as assignment syntax. In the premiss we use := for assigns.

Assignment When an assignment is used, we can extract the following constraint: the right hand side (E_2) of the assignment is a subtype of the left hand side (E_1). This relation is a subtype relation, not an is equal relation, because of the subclass relations of inheritance. The whole expression (E) is equal to the newly assigned value.

$$\frac{E \equiv (E_1 := E_2)}{\begin{array}{c}[\![E_2]\!] <: [\![E_1]\!], \\ [\![E_1]\!] = [\![E]\!]\end{array}}$$

Code sample:

```
$a = $b;       // [$b] <: [$a]
               // [$a] = [$a = $b]

$c = $d = $e; // [$e] <: [$d]
               // [$d] <: [$c],
               // [$d] = [$d = $e]
               // [$c] = [$c = $d = $e]
```

Listing 4.6: Assignment

Ternary operator The *ternary* operator is a conditional assignment. If the expression E_1 is evaluated as `true`, the left hand side (E_2) is the value of the whole ternary expression (i). If E_1 is evaluated as `false`, the right hand side (E_3) is the value. The constraint we can extract from the ternary expression is that the type of the whole expression should be the type of E_2 or E_3.
The ternary operator without a left hand side value (ii), also known as the elvis operator, returns the value of E_1 when E_1 is evaluated as `true`. Here the type of the expression should be either the type of E_1 or E_3.

$$\frac{E \equiv (E_1 \ ? \ E_2 : E_3)}{[\![E]\!] = [\![E_2]\!] \vee [\![E_3]\!]} \text{ (i)} \qquad \frac{E \equiv (E_1 \ ? : E_3)}{[\![E]\!] = [\![E_1]\!] \vee [\![E_3]\!]} \text{ (ii)}$$

Code sample:

```
$a ? $b : $c; // [$a ? $b : $c] = ([$b] || [$c])
$a ?: $c;     // [$a ?: $c]     = ([$a] || [$c])
```

Listing 4.7: Ternary operator

Assignments resulting in integers PHP provides several assignment statements combined with operators. The type of the left hand side (E_1) is in the cases of *bitwise and* (i), *bitwise inclusive or* (ii), *bitwise exclusive or* (iii), *bitwise shift left* (iv), *bitwise shift right* (v), and *modulus* (vi) always of *integer* type.

$$\frac{E_1 \ \& = E_2}{[\![E_1]\!] = \{ \text{ integerType() } \}} \text{ (i)} \qquad \frac{E_1 \ | = E_2}{[\![E_1]\!] = \{ \text{ integerType() } \}} \text{ (ii)} \qquad \frac{E_1 \ \hat{} = E_2}{[\![E_1]\!] = \{ \text{ integerType() } \}} \text{ (iii)}$$

$$\frac{E_1 <<= E_2}{[\![E_1]\!] = \{ \text{ integerType() } \}} \text{ (iv)} \qquad \frac{E_1 >>= E_2}{[\![E_1]\!] = \{ \text{ integerType() } \}} \text{ (v)} \qquad \frac{E_1 \ \% = E_2}{[\![E_1]\!] = \{ \text{ integerType() } \}} \text{ (vi)}$$

Code sample:

```
$a  &= $b; // [$a] = integerType()
$a  |= $b; // [$a] = integerType()
$a  ^= $b; // [$a] = integerType()
$a <<= $b; // [$a] = integerType()
$a >>= $b; // [$a] = integerType()
$a  %= $b; // [$a] = integerType()
```

Listing 4.8: Assignments resulting in integers

Assignment with string concat When the *string concat* operator is used, in combination with the assignment operator (i), the type of the left hand side (E_1) is always a string.
About the right hand side (E_2) we can say that **if** the type of E_2 is a subtype of object, then this object should have the method *__toString*() (ii).

$$\frac{E_1 \ .= E_2}{[\![E_1]\!] = \{ \text{ stringType() } \}} \text{ (i)} \qquad \frac{(E_1 \ .= E_2) \qquad ([\![E_2]\!] <: \{ \text{ objectType() } \})}{[\![E_2]\!] \ \texttt{hasMethod} \ \text{"}\_\_tostring\text{"}} \text{ (ii)}$$

Code sample:

```
$a .= $b;  // [$a] = stringType()
// An error occurs when $b is of type object() and
// __toString is not defined or does not return a string
```

Listing 4.9: Assignment with string concat

Assignments with division or subtraction operator *Division* (i) and *subtraction* (ii) assignment in PHP will always result in an integer type. This is the case for all values, except for array's. A fatal error will occur when the right hand side value is of type array.

$$\frac{E_1 \ /= E_2}{\begin{array}{c}[\![E_1]\!] = \{\ \text{integerType}()\ \},\\ [\![E_2]\!] \neq \{\ \text{arrayType}(\_)\ \}\end{array}}\ \text{(i)} \qquad \frac{E_1 -= E_2}{\begin{array}{c}[\![E_1]\!] = \{\ \text{integerType}()\ \},\\ [\![E_2]\!] \neq \{\ \text{arrayType}(\_)\ \}\end{array}}\ \text{(ii)}$$

Code sample:

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

Listing 4.10: Assignments with division or subtraction operator

Assignments resulting in numbers The result of an *multiplication* (i) and *addition* (ii) assignment is either a float or an integer. When the type of the right hand side (E_2) is either booleanType, integerType, or nullType, the result of the assignment (E_1) will be of integerType. If E_2 is of any other type, E_1 will be of type floatType. Float and integer are both subtypes of integers, so we can use the subtype relation for numberType for this.

$$\frac{E_1 \ *= E_2}{[\![E_1]\!] <: \{\ \text{numberType}()\ \}}\ \text{(i)} \qquad \frac{E_1 \ += E_2}{[\![E_1]\!] <: \{\ \text{numberType}()\ \}}\ \text{(ii)}$$

Code sample:

```
$a *= $b; // [$a] <: numberType()
$a += $b; // [$a] <: numberType()
```

Listing 4.11: Assignments resulting in numbers

4.1.3 Unary operators

Unary operators in PHP consist of positive and negative numbers, negation operators, and increase and decrease operators.

Positive and negative numbers When a *plus* (i) or *minus* (ii) sign is used in PHP in front of a variable, the type of the whole expression must be of numberType. The type of the variable cannot be of any arrayType.

$$\frac{E \equiv (+E_1)}{\begin{array}{c}[\![E]\!] <: \{\ \text{numberType}()\ \},\\ [\![E_1]\!] \neq \{\ \text{arrayType}(\_)\ \}\end{array}}\ \text{(i)} \qquad \frac{E \equiv (-E_1)}{\begin{array}{c}[\![E]\!] <: \{\ \text{numberType}()\ \},\\ [\![E_1]\!] \neq \{\ \text{arrayType}(\_)\ \}\end{array}}\ \text{(ii)}$$

Code sample:

```
+$a; // [$a] <: numberType();
     // [$a] =/= arrayType();
-$a; // [$a] <: numberType();
     // [$a] =/= arrayType();
```

Listing 4.12: Positive and negative numbers

Negation operators The PHP language holds two types of negation operators. The type of the whole expression for *normal negation* operator (i) is boolean. For the *bitwise negation* operator (ii) the type of attached variable is either a number or a string. The type of the whole expression is an integer or string.

$$\frac{E \equiv\ (!E_1)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(i)} \qquad \frac{E \equiv\ (\sim E_1)}{\begin{array}{c}[\![E_1]\!] = \{\ \text{numberType(), stringType()}\ \},\\ [\![E]\!] = \{\ \text{integerType(), stringType()}\ \}\end{array}}\ \text{(ii)}$$

Code sample:

```
!$a // [!$a] = booleanType()
~$a // [$a] = numberType() or stringType()
    // [~$a] = integerType() or stringType()
```

Listing 4.13: Negation operators

Post increment operators From post increment and decrement operators we can only extract conditional constraints.
If the type of E_1 is of any `array` type, the result of the expression is also of any `array` type (i).
If the type of E_1 is of `boolean` type, the result of the expression is also of `boolean` type (ii).
If the type of E_1 is of `float` type, the result of the expression is also of `float` type (iii).
If the type of E_1 is of `integer` type, the result of the expression is also of `integer` type (iv).
If the type of E_1 is of `null` type, the result of the expression is either of `integer` or `boolean` type (v).
If the type of E_1 is of any `object` type, the result of the expression is also of any `object` type (vi).
If the type of E_1 is of `resource` type, the result of the expression is also of any `resource` type (vii).
If the type of E_1 is of `string` type, the result of the expression is either of `number` or `string` type (vii).
The rules below are only written for the post increment, but also apply on the post decrement.

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] <: \{\ \text{arrayType(\_)}\ \})}{[\![E]\!] <: \{\ \text{arrayType(\_)}\ \}}\ \text{(i)}$$

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] = \{\ \text{booleanType()}\ \})}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(ii)}$$

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] = \{\ \text{floatType()}\ \})}{[\![E]\!] = \{\ \text{floatType()}\ \}}\ \text{(iii)}$$

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] = \{\ \text{integerType()}\ \})}{[\![E]\!] = \{\ \text{integerType()}\ \}}\ \text{(iv)}$$

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] = \{\ \text{nullType()}\ \})}{[\![E]\!] = \{\ \text{integerType(), nullType()}\ \}}\ \text{(v)}$$

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] <: \{\ \text{objectType()}\ \})}{[\![E]\!] <: \{\ \text{objectType()}\ \}}\ \text{(vi)}$$

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] = \{\ \text{resourceType()}\ \})}{[\![E]\!] = \{\ \text{resourceType()}\ \}}\ \text{(vii)}$$

$$\frac{(E \equiv (E_1 ++)) \qquad ([\![E_1]\!] = \{\ \text{stringType()}\ \})}{[\![E]\!] <: \{\ \text{numberType(), stringType()}\ \}}\ \text{(viii)}$$

Code sample:

```
$a++ // (post increase)
     // if ([$a] <: arrayType())    => [$a++] <: arrayType()
     // if ([$a]  = booleanType())  => [$a++]  = booleanType()
     // if ([$a]  = floatType())    => [$a++]  = floatType()
     // if ([$a]  = integerType())  => [$a++]  = integerType()
     // if ([$a]  = nullType())     => [$a++]  = integerType() or nullType()
     // if ([$a] <: objectType())   => [$a++] <: objectType()
     // if ([$a]  = resourceType()) => [$a++]  = resourceTYpe()
     // if ([$a]  = stringType())   => [$a++] <: numberType() or stringType()
$a-- // (post decrease)
     // same rules as above apply for $a--
```

Listing 4.14: Post increment operators

Pre increment operators From pre increment and decrement operators we can also only extract conditional constraints. The rules are similar to the rules for the post increment, except for the `nullType()`. If the type of E_1 is of `null` type, the result of the expression is either of `null` type (v).

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] <: \{\ \text{arrayType(\_)}\ \})}{[\![E]\!] <: \{\ \text{arrayType(\_)}\ \}}\ \text{(i)}$$

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] = \{\ \text{booleanType()}\ \})}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(ii)}$$

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] = \{\ \text{floatType()}\ \})}{[\![E]\!] = \{\ \text{floatType()}\ \}}\ \text{(iii)}$$

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] = \{\ \text{integerType()}\ \})}{[\![E]\!] = \{\ \text{integerType()}\ \}}\ \text{(iv)}$$

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] = \{\ \text{nullType()}\ \})}{[\![E]\!] = \{\ \text{nullType()}\ \}}\ \text{(v)}$$

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] <: \{\ \text{objectType()}\ \})}{[\![E]\!] <: \{\ \text{objectType()}\ \}}\ \text{(vi)}$$

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] = \{\ \text{resourceType()}\ \})}{[\![E]\!] = \{\ \text{resourceType()}\ \}}\ \text{(vii)}$$

$$\frac{(E \equiv (++E_1)) \qquad ([\![E_1]\!] = \{\ \text{stringType()}\ \})}{[\![E]\!] <: \{\ \text{numberType(), stringType()}\ \}}\ \text{(viii)}$$

Code sample:

```
++$a // (pre increase)
     // if ([$a] <: arrayType())    => [++$a] <: arrayType()
     // if ([$a]  = booleanType())  => [++$a]  = booleanType()
     // if ([$a]  = floatType())    => [++$a]  = floatType()
     // if ([$a]  = integerType())  => [++$a]  = integerType()
     // if ([$a]  = nullType())     => [++$a]  = nullType()
     // if ([$a] <: objectType())   => [++$a] <: objectType()
     // if ([$a]  = resourceType()) => [++$a]  = resourceTYpe()
     // if ([$a]  = stringType())   => [++$a] <: numberType() or stringType()
--$a // (pre decrease)
     // same rules as above apply for --$a
```

Listing 4.15: Pre increment operators

4.1.4 Binary operators

Addition-, subtraction-, multiplication-, division-, modulus-, bitwise-, comparison-, and logical operators are in PHP binary operators.

Addition operator The result of an addition operator will always be a number or an array (i). If the left and right hand side are both arrays, the return type will be array (ii). In this case two arrays are merged. In all other cases the result of this operation is a number (iii).

$$\frac{E \equiv (E_1 + E_2)}{[\![E]\!] <: \{\ \text{arrayType}(\_),\ \text{numberType}()\ \}}\ \text{(i)}$$

$$\frac{E \equiv (E_1 + E_2) \qquad [\![E_1]\!] <: \{\ \text{arrayType}(\_)\ \} \wedge [\![E_2]\!] <: \{\ \text{arrayType}(\_)\ \}}{[\![E]\!] <: \{\ \text{arrayType}(\_)\ \}}\ \text{(ii)}$$

$$\frac{E \equiv (E_1 + E_2) \qquad [\![E_1]\!]\ !<: \{\ \text{arrayType}(\_)\ \} \vee [\![E_2]\!]\ !<: \{\ \text{arrayType}(\_)\ \}}{[\![E]\!] <: \{\ \text{numberType}()\ \}}\ \text{(iii)}$$

Code sample:

```
$a + $b // (addition)
        // [$a + $b] <: arrayType() or numberType()
        // if (([$a] and [$b])  <: arrayType(_)) => [$a + $b] <: arrayType(_)
        // if (([$a] or  [$b]) !<: arrayType(_)) => [$a + $b] <: numberType()
```

Listing 4.16: Addition operator

Subtraction multiplication division operators The *subtraction* (i), *multiplication* (ii), and *division* (iii) operators are merged together in this paragraph because they have identical behaviour. The result of these operations is always of `number` type. The operations cannot be used if one of the sides is of type `array`. Therefore we can says that the left and right hand side cannot be of `array` type.

$$\frac{E \equiv (E_1 - E_2)}{\begin{array}{c}[\![E]\!] <: \{\ \text{numberType}()\ \},\\ [\![E_1]\!]\ !<: \{\ \text{arrayType}(\_)\ \},\\ [\![E_2]\!]\ !<: \{\ \text{arrayType}(\_)\ \}\end{array}}\ \text{(i)} \qquad \frac{E \equiv (E_1 * E_2)}{\begin{array}{c}[\![E]\!] <: \{\ \text{numberType}()\ \},\\ [\![E_1]\!]\ !<: \{\ \text{arrayType}(\_)\ \},\\ [\![E_2]\!]\ !<: \{\ \text{arrayType}(\_)\ \}\end{array}}\ \text{(ii)}$$

$$\frac{E \equiv (E_1 / E_2)}{\begin{array}{l}[\![E]\!] <: \{\ \text{numberType}()\ \},\\ [\![E_1]\!]\ !<: \{\ \text{arrayType}(\_)\ \},\\ [\![E_2]\!]\ !<: \{\ \text{arrayType}(\_)\ \}\end{array}}\ \text{(iii)}$$

Code sample:

```
$a - $b // (subtraction)
$a * $b // (mulitiplication)
$a / $b // (division)
        // [$a - $b] <: numberType()
        // [$a * $b] <: numberType()
        // [$a / $b] <: numberType()
        // [$a] !<: array(_)
        // [$b] !<: array(_)
```

Listing 4.17: Subtraction multiplication division operators

Modulus and bitwise shift operators The merge of *modulus* (i) and *bitwise shift* (ii, iii) operators seems not so obvious at first, but they have the same behaviour. The results of these operations is of integer type.

$$\frac{E \equiv (E_1\ \%\ E_2)}{[\![E]\!] = \{\ \text{integerType}()\ \}}\ \text{(i)} \qquad \frac{E \equiv (E_1 << E_2)}{[\![E]\!] = \{\ \text{integerType}()\ \}}\ \text{(ii)} \qquad \frac{E \equiv (E_1 >> E_2)}{[\![E]\!] = \{\ \text{integerType}()\ \}}\ \text{(iii)}$$

Code sample:

```
$a % $b  // [$a % $b]  = integerType()  // (modulus)
$a << $b // [$a << $b] = integerType()  // (bitwise shift left)
$a >> $b // [$a >> $b] = integerType()  // (bitwise shift right)
```

Listing 4.18: Modulus and bitwise shift operators

Bitwise operators The results of the bitwise operators *and* (i, ii, iii), *or*, and *xor* is always of `integer` or `string` type. When the left and right hand side are both strings, the result of the operation is also of type `string`. In all other cases the result of this operation is a number.

$$\frac{E \equiv (E_1\ \&\ E_2)}{[\![E]\!] = \{\ \text{stringType}(), \text{integerType}()\ \}}\ \text{(i)}$$

$$\frac{E \equiv (E_1\ \&\ E_2) \qquad [\![E_1]\!] = \{\ \text{stringType}()\ \} \wedge [\![E_2]\!] = \{\ \text{stringType}()\ \}}{[\![E]\!] = \{\ \text{stringType}()\ \}}\ \text{(ii)}$$

$$\frac{E \equiv (E_1\ \&\ E_2) \qquad [\![E_1]\!] \neq \{\ \text{stringType}()\ \} \vee [\![E_2]\!] \neq \{\ \text{stringType}()\ \}}{[\![E]\!] = \{\ \text{integerType}()\ \}}\ \text{(iii)}$$

Code sample:

```
$a & $b // (bitwise And)
        // [$a & $b] = stringType() or integerType()
        // if (($a and $b) = stringType()) => [$a & $b] = stringType()
```

```
        // if (($a or $b) != stringType()) => [$a & $b] = integerType()
$a | $b // (bitwise Or)
        // [$a | $b] = stringType() or integerType()
        // if (($a and $b) = stringType()) => [$a | $b] = stringType()
        // if (($a or $b) != stringType()) => [$a | $b] = integerType()
$a ^ $b // (bitwise Xor)
        // [$a ^ $b] = stringType() or integerType()
        // if (($a and $b) = stringType()) => [$a ^ $b] = stringType()
        // if (($a or $b) != stringType()) => [$a ^ $b] = integerType()
```

Listing 4.19: Bitwise operators

Comparison operators The result of the comparison operators is always of boolean type. The comparison operators are *equals* (i), *identical* (ii), *not equal* (iii), *not equal* (iv), *not identical* (v), *less than* (vi), *greater than* (vii), *less than or equal to* (viii), and *greater than or equal to* (ix) operators.

$$\frac{E \equiv (E_1 == E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(i)} \qquad \frac{E \equiv (E_1 === E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(ii)} \qquad \frac{E \equiv (E_1\ != E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(iii)}$$

$$\frac{E \equiv (E_1 <> E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(iv)} \qquad \frac{E \equiv (E_1\ !== E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(v)} \qquad \frac{E \equiv (E_1 < E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(vi)}$$

$$\frac{E \equiv (E_1 > E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(vii)} \qquad \frac{E \equiv (E_1 <= E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(viii)}$$

$$\frac{E \equiv (E_1 >= E_2)}{[\![E]\!] = \{\ \text{booleanType()}\ \}}\ \text{(ix)}$$

Code sample:

```
$a == $b  // [$a == $b]  = booleanType()
$a === $b // [$a === $b] = booleanType()
$a != $b  // [$a != $b]  = booleanType()
$a <> $b  // [$a <> $b]  = booleanType()
$a !== $b // [$a !== $b] = booleanType()
$a < $b   // [$a < $b]   = booleanType()
$a > $b   // [$a > $b]   = booleanType()
$a <= $b  // [$a <= $b]  = booleanType()
$a >= $b  // [$a >= $b]  = booleanType()
```

Listing 4.20: Comparison operators

Logical operators Just like the comparison operators, the result of the logical operators is always of boolean type. The logical operators are *and* (i), *or* (ii), *xor* (iii), *and* (iv), and *or* (v).

$$\frac{E \equiv (E_1\ and\ E_2)}{[E] = \{\ \text{booleanType()}\ \}}\ \text{(i)} \qquad \frac{E \equiv (E_1\ or\ E_2)}{[E] = \{\ \text{booleanType()}\ \}}\ \text{(ii)} \qquad \frac{E \equiv (E_1\ xor\ E_2)}{[E] = \{\ \text{booleanType()}\ \}}\ \text{(iii)}$$

$$\frac{E \equiv (E_1\ \&\&\ E_2)}{[E] = \{\ \text{booleanType()}\ \}}\ \text{(iv)} \qquad \frac{E \equiv (E_1\ ||\ E_2)}{[E] = \{\ \text{booleanType()}\ \}}\ \text{(v)}$$

Code sample:

```
$a and $b // [$a and $b] = booleanType()
$a or $b  // [$a or $b]  = booleanType()
$a xor $b // [$a xor $b] = booleanType()
$a && $b  // [$a && $b]  = booleanType()
$a || $b  // [$a || $b]  = booleanType()
```

Listing 4.21: Logical operators

4.1.5 Array

From the PHP parser we get array declaration and array access nodes.

Array declaration From the array declarations (`array()` or `[]`) we can extract the constraint that they should be of any `array` type.

$$\frac{E \texttt{ is\_a } \text{array}}{[\![E]\!] <: \{ \text{ arrayType(\_) } \}}$$

Code sample:

```
array(/*..*/); // [array(/*..*/)] <: arrayType(any())
[/*..*/];      // [[(/*..*/)]]    <: arrayType(any())
```

Listing 4.22: Array declaration

Array access From the usage of array access syntax you cannot tell what the type of the expression is. The same syntax is used to access strings. We can extract that the type of the base expression should not be of object type (i). If we know that the base type is of `string` type, we know that the result of the expression will also be a string (ii). When the base type is an array, the result type is the type of the elements in there array (iii). For all other cases, when the base type is not an string or array, the result of the expression will be of `null` type (iv).

$$\frac{E_1[E_2] \qquad E_1 \texttt{ is\_a } \text{arrayAccess}}{[\![E_1]\!] \neq \{ \text{ objectType() } \}} \text{ (i)}$$

$$\frac{E \equiv (E_1[E_2]) \qquad E_1 \texttt{ is\_a } \text{arrayAccess} \qquad [\![E_1]\!] = \{ \text{ stringType() } \}}{[\![E]\!] = \{ \text{ stringType() } \}} \text{ (ii)}$$

$$\frac{E \equiv (E_1[E_2]) \qquad E_1 \texttt{ is\_a } \text{arrayAccess} \qquad [\![E_1]\!] = \{ \text{ arrayType}(E_2) \}}{[\![E]\!] = [\![E_2]\!]} \text{ (iii)}$$

$$\frac{E \equiv (E_1[E_2]) \qquad E_1 \texttt{ is\_a } \text{arrayAccess} \qquad [\![E_1]\!] \neq \{ \text{ stringType() } \} \qquad [\![E_1]\!] \,!<: \{ \text{ arrayType(\_) } \}}{[\![E]\!] = \{ \text{ nullType() } \}} \text{ (iv)}$$

Code sample:

```
$a[$b];
// [$a] != objectType()
// if ([$a] == stringType())      => [$a[$b]] = stringType()
// if ([$a] == arrayType(x)       => [$a[$b]] = [x]
// if ([$a] != (string or array) => [$a[$b]] = nullType()
```

Listing 4.23: Array access

4.1.6 Casts

Casts PHP contains syntax to arrays, booleans, integers, floats, objects, strings, and to unset variables. The result of a cast to array is of any `array` type (i). For casting to boolean there are two keywords, *bool* (ii) and *boolean* (iii), and the result will always be of `boolean` type. There are three keywords to cast to floats, *float* (iv), *double* (v), and *real* (vi). Casts to integer integer type, you can use *integer* (vii) or *int* (viii) keywords. Any cast to *object* (ix) will result in any `object` type. A cast to *string* will always result in a `string` type. String casts (x) will always result in a `string` type. If we know that the expression (E_1) is an object, we know that this method needs to have an *__toString*() method (xi). The last cast, *unset*, results in a `null` type.

$$\frac{E \equiv (array)E_1}{[\![E]\!] <: \{\ \text{arrayType}(\_)\ \}}\ \text{(i)} \qquad \frac{E \equiv (\texttt{boolean})E_1}{[\![E]\!] = \{\ \text{booleanType}()\ \}}\ \text{(ii)} \qquad \frac{E \equiv (\texttt{bool})E_1}{[\![E]\!] = \{\ \text{booleanType}()\ \}}\ \text{(iii)}$$

$$\frac{E \equiv (\texttt{float})E_1}{[\![E]\!] = \{\ \text{floatType}()\ \}}\ \text{(iv)} \qquad \frac{E \equiv (\texttt{double})E_1}{[\![E]\!] = \{\ \text{floatType}()\ \}}\ \text{(v)} \qquad \frac{E \equiv (\texttt{real})E_1}{[\![E]\!] = \{\ \text{floatType}()\ \}}\ \text{(vi)}$$

$$\frac{E \equiv (\texttt{integer})E_1}{[\![E]\!] = \{\ \text{integerType}()\ \}}\ \text{(vii)} \qquad \frac{E \equiv (\texttt{int})E_1}{[\![E]\!] = \{\ \text{integerType}()\ \}}\ \text{(viii)} \qquad \frac{E \equiv (\texttt{object})E_1}{[\![E]\!] <: \{\ \text{objectType}()\ \}}\ \text{(ix)}$$

$$\frac{E \equiv (\texttt{string})E_1}{[\![E_1]\!] = \{\ \text{stringType}()\ \}}\ \text{(x)} \qquad \frac{E \equiv (\texttt{string})E_1 \qquad (E_1 <: \{\ \text{objectType}()\ \})}{[\![E_1]\!]\ \texttt{hasMethod}\ \text{"}\_\_tostring\text{"}}\ \text{(xi)}$$

$$\frac{E \equiv (unset)E_1}{[\![E]\!] = \{\ \text{nullType}()\ \}}\ \text{(xii)}$$

Code sample:

```
(array)$a  // [(array)$a]  <: arrayType(_)
(bool)$a   // [(bool)$a]   = booleanType()
(float)$a  // [(float)$a]  = floatType()
(int)$a    // [(int)$a]    = integerType()
(object)$a // [(object)$a] = objectType()
(string)$a // [(string)$a] = stringType()
           // if ($a <: objectType()) => [$a] has method "__toString()"
(unset)$a  // [(unset)$a]  = nullType()
```

Listing 4.24: Casts

4.1.7 Clone

Clone From the PHP function *clone* we can extract the constraint that the type of the given expression and the result must be of any `object` type. We also know that the type will not change, and so the type of the expression will be the same as

$$\frac{E \equiv \texttt{clone}(E_1)}{\begin{array}{c}[\![E]\!] <: \{\ \text{objectType}()\ \} \\ [\![E_1]\!] <: \{\ \text{objectType}()\ \} \\ [\![E]\!] = [\![E_1]\!]\end{array}}$$

Code sample:

```
clone($a) // [$a]        <: object
          // [clone($a)] <: object
          // [$a]        =  [clone($a)]
```

Listing 4.25: Clone

4.1.8 Class

This section contains fact extraction rules from object syntax. Class instantiation, special keywords, method calls, parameters, and class constants.

Class instantiation Classes can be instantiated with the name of the class. The type of the whole expression is then of the specific `class` type (i). When a class is dynamically instantiated, we only know that it should be of some `object` type, and that the type of the expression should be any `object` or `string` type (ii).

$$\frac{E \equiv \mathit{new}\ C_1()}{[\![E]\!] = \{\ \text{classType(C.decl)}\ \}}\ \text{(i)} \qquad \frac{E \equiv \mathit{new}\ E_1}{\begin{array}{c}[\![E]\!] <: \{\ \text{objectType()}\ \}, \\ [\![E_1]\!] <: \{\ \text{objectType(), stringType()}\ \}\end{array}}\ \text{(ii)}$$

Code sample:

```
new C;     // [new C] = classType(C)

$c = "C";
new $c;    // [new $c] <: objectType()
           // [$c]     <: ( objectType() or stringType() )
```

Listing 4.26: Class instantiation

Special keywords PHP contains a few class related reserved keywords with special behaviour. These keywords can be used inside a class scope ($\in C$). From the usage of the keyword *self* we know that the type of the expression should be the same `class` type as which the keyword is defined in (i). The constraint we can extract from *self* is that the type should be any `object` type and it should be either the contained class or one of the parent classes. The behaviour of *$this* (ii) and *static* (iii) differs, but the constraints we can extract are equal to the *self* keyword. The *parent* (iv) keyword differs because it must be a super type of the class they keyword is defined in.

$$\frac{(E \equiv \texttt{self}) \in C}{\begin{array}{c}[\![E]\!] <: \{\ \text{objectType()}\ \}, \\ [\![E]\!] = \text{classType(C)} \lor [\![E]\!] :> \text{classType(C)}\end{array}}\ \text{(i)}$$

$$\frac{(E \equiv \texttt{static}) \in C}{\begin{array}{c}[\![E]\!] <: \{\ \text{objectType()}\ \}, \\ [\![E]\!] = \text{classType(C)} \lor [\![E]\!] :> \text{classType(C)}\end{array}}\ \text{(ii)}$$

$$\frac{(E \equiv \texttt{\$this}) \in C}{\begin{array}{c}[\![E]\!] <: \{\ \text{objectType()}\ \}, \\ [\![E]\!] = \text{classType(C)} \lor [\![E]\!] :> \text{classType(C)}\end{array}}\ \text{(iii)}$$

$$\frac{(E \equiv \texttt{parent}) \in C}{\begin{array}{c}[\![E]\!] <: \{\ \text{objectType()}\ \},\\ [\![E]\!] :> \text{classType(C)}\end{array}}\ \text{(iv)}$$

Code sample:

```
self   // in class C -> [self]   = classType(C)
parent // in class C -> [parent] = parentOf(classType(C))
static // in class C -> [static] = classType(C) or <: classType(C)
$this  // in class C -> [$this]  = classType(C) or <: classType(C)
```

Listing 4.27: Special keywords

Method calls From the usage of a method call (*expression* -> *expression*) we can extract the constraint that the type of the left hand side should be an object (i and ii). If the right hand side (E_2) is a name of a method, we can extract the constraint that the left hand side (E_1) must implement this method (ii).

$$\frac{E_1 \rightarrow E_2 \in C \qquad E_2\ \texttt{is\_a}\ \text{expression}}{[\![E_1]\!] <: \text{objectType()}}\ \text{(i)}$$

$$\frac{E_1 \rightarrow E_2 \in C \qquad E_2\ \texttt{is\_a}\ \text{name}}{\begin{array}{c}[\![E_1]\!] <: \text{objectType()},\\ [\![E_1]\!] = C.\text{hasMethod}(E_2.\text{name, static} \notin \text{Mfs})\end{array}}\ \text{(ii)}$$

Code sample:

```
$a->$b() // [$a] <: objectType()
$a->b()  // [$a] <: objectType()
         // [$a] has a method (possible inherited) with the name 'b'
```

Listing 4.28: Method calls

4.1.9 Scope

In order to define the type of a variable within a scope, we have conducted the following constraints.

Variables The type of a certain variable is defined by adding an equal constraint on the logical name and the location. The scope of these variables is contained in the logical name. The logical name contains the name of the scope, which is optionally a namespace and the name of a class, method or function, and the name of the variable. An example of a logical name for the variable `$a` in the function `f` in the global namespace is `|php+functionVar:///f/a`.

$$\frac{E \qquad E\ \text{is\_a variable}}{[\![E_{definition}]\!] = [\![E_{loction}]\!]}$$

Code sample:

```
function f() {
  $a = 1;    // [|php+functionVar:///f/a|] = [|file:///file.php|(17,2,<2,0>,<2,0>))]
  $a = "s"; // [|php+functionVar:///f/a|] = [|file:///file.php|(27,2,<3,0>,<3,0>]
}
```

Listing 4.29: Variables

Return types The type of a function or method is defined by the return statements it contains. When there are no return statements declared in a function or method (i), we can extract the constraint that the function will always return `nullType`. The type of a function is also `nullType` when there is a return statement without an expression (ii), like `return;`. When there are multiple return statements, the return type of the function or method is the concatenation of the types of the expressions (iii).

$$\frac{\text{E } \texttt{is\_a} \text{ return} \not\subseteq f}{[\![f]\!] = nullType()} \text{(i)} \qquad \frac{\texttt{is\_a} \text{ return E} \subseteq \text{f} \qquad \text{E } \texttt{is\_a} \text{ noExpr}}{[\![f]\!] = nullType()} \text{(ii)}$$

$$\frac{(\text{return } E_1) \vee \cdots \vee (\text{return } E_k) \subseteq f \qquad E_{1\ldots k} \texttt{ is\_a} \text{ someExpr}}{[\![f]\!] <: [\![E_1]\!] \vee \cdots \vee [\![E_k]\!]} \text{(iii)}$$

Code sample:

```
function f() {}           // [f] = nullType()
function f() { return; } // [f] = nullType()

function f() {            // [f] = [$a] or [$b]
  return mt_rand(0,1) ? $a : $b;
}
```

Listing 4.30: Return types

4.2 Annotations

Annotations are pieces of meta data, defined on class, method, function, or statement level. Despite the proposal[1]for official support of annotations, PHP has still no native support for them. But PHP has a `getDocComment`[2]method in the `ReflectionClass` since version 5.1 in 2005. The `getDocComment` method returns the complete doc block of a certain element as a string. A doc block in php has the format `/**...*/`. Listing 4.31 shows an example of two doc blocks in PHP. The first doc block is defined above the class and contains information about the class. The second doc block is related to the method `getSomething`. The block contains a short description of the method, provides type hints for the parameter and the return type, and provides information which possible exceptions can be thrown by the method.

```
namespace Thesis;

/**
 * Class Example
 * @package Thesis
 */
class Example
{
    /**
     * This is a description of the method getSomething
     *
     * @param SomeTypeHint $someObject
     * @return string
     * @throws NoNameException
     */
    public function getSomething(SomeTypeHint $someObject)
```

[1]https://wiki.php.net/rfc/annotations-in-docblock
[2]http://php.net/manual/en/reflectionclass.getdoccomment.php

```
17     {
18         if (null === $someObject->getName()) {
19             throw new NoNameException();
20         }
21
22         return $someObject->getName();
23     }
24 }
```

Listing 4.31: Examples of PHP DocBlocks

Annotations are mainly used for type hinting, documentation, and code execution. Software analysis tools and IDE's can use the type hints to aid understanding code and in finding bugs and security issues. Available tools can generate documentation based on the doc blocks. Programs like Symfony2, ZEND Framework, and Doctrine ORM use annotations for controller routing, templating information, ORM mappings, filters, and validation configuration.
This research focuses on the first type of annotations which can help developers and IDE's to better understand how code behaves within a program. For example a programmer can see what kind of input and output is expected for a method. Doc blocks with annotations can be placed on top of classes, methods, functions, and variables.
A standard on using annotations is not in the PHP Standard Recommendations (PSR) yet, but there is a proposal[3]. 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, class methods, and closures. 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[4] syntax. For this research the following annotations are considered:

$$\texttt{@return} = \begin{cases} \texttt{@return}\ \textit{type}, & \text{unconditionally read } \texttt{@return}\ \textit{type}. \end{cases} \tag{4.1}$$

$$\texttt{@param} = \begin{cases} \texttt{@param}\ \textit{type}\ \$\textit{var}, & \text{if '}\texttt{@param}\ \textit{type}\ \$\text{var' occurs at least once.} \\ \texttt{@param}\ \$\textit{var}\ \textit{type}, & \text{else if '}\texttt{@param}\ \$\text{var}\ \textit{type}\text{' occurs at least once.} \\ \texttt{@param}\ \textit{type}, & \text{otherwise try to match '}\texttt{@param}\ \textit{type}\text{'.} \end{cases} \tag{4.2}$$

$$\texttt{@var} = \begin{cases} \texttt{@var}\ \textit{type}\ \$\textit{var}, & \text{if '}\texttt{@var}\ \textit{type}\ \$\text{var' occurs at least once.} \\ \texttt{@var}\ \$\textit{var}\ \textit{type}, & \text{else if '}\texttt{@var}\ \$\text{var}\ \textit{type}\text{' occurs at least once.} \\ \texttt{@var}\ \textit{type}, & \text{otherwise try to match '}\texttt{@var}\ \textit{type}\text{'.} \end{cases} \tag{4.3}$$

$$\textit{type} = \begin{cases} \textit{type}|\textit{type}, & \text{if '|' in } \textit{type}. \\ \textit{type}, & \text{otherwise} \end{cases} \tag{4.4}$$

In equation 4.1 we show the supported syntax for return annotations. This format starts with the `@return` annotation followed by the type hint. The return type hint is used on functions and class methods to provide information about the possible return values.

In equation 4.2 we describe the variants of `@param` annotations. Because the usage is not standardised we support multiple variants. The first variant has a type hint and next provides information for which variable. Next we also support the typehint and variable to be swapped.

In equation 4.3 we show the variants for the `@var` annotation. They have the same support as the `@param` annotation.

In equation 4.4 we show the recursive definition of a type. This is a recursive definition because one type hint can have multiple types. When there are multiple return types a | is used.

[3] https://github.com/php-fig/fig-standards/pull/169/files, July 2014
[4] http://www.phpdoc.org/

4.3 PHP built-ins

Native functions are a part of PHP built-ins. Predefined classes, interfaces, constants, and variables are also part of PHP built-ins. We have included the PHP built-in information to see if we can improve the analysis results. To be able analyse the PHP built-ins we've included the PHP representation it. These files, written in PHP, provide annotation on what types flow in and out the functions. A brief example is shown in listing 4.32. From this example we can fetch the parameter and return type of the `strtoupper` and `strtolower` functions.

```
/**
 * Make a string uppercase
 * @link http://php.net/manual/en/function.strtoupper.php
 * @param string $string <p>
 * The input string.
 * </p>
 * @return string the uppercased string.
 * @since 4.0
 * @since 5.0
 */
function strtoupper ($string) {}

/**
 * Make a string lowercase
 * @link http://php.net/manual/en/function.strtolower.php
 * @param string $str <p>
 * The input string.
 * </p>
 * @return string the lowercased string.
 * @since 4.0
 * @since 5.0
 */
function strtolower ($str) {}
```

Listing 4.32: Short samples of built-in PHP file

Chapter 5

Implementation of PHP type inference

The type inference is executed in 3 main steps. The first step creates an M^3 model for a PHP program which contains various facts about the program. The creation of an M^3 for PHP is explained in more details in section 5.1. Once the M^3 model is constructed, we extract constraints from the program using the M^3 model. Next, as explained in section 5.3, we solve the constraints and resolve the types of variables used in the program. The final section 5.4 of this chapter explains how annotations and PHP built-in information is included in the process to improve the type inference.

5.1 M^3 for PHP

As explained in section 2.2, 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 5.1. Independent M^3 models are build each PHP file in the program.

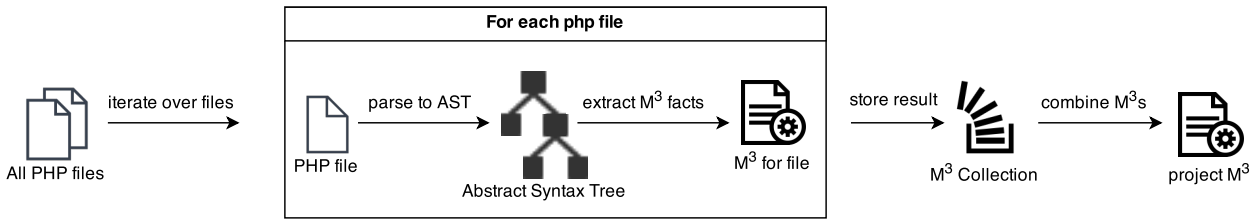


Figure 5.1: M^3 Creation

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.

5.1.1 Core elements

The M^3 model has the following core elements: `declarations`, `containment`, `modifiers`, `uses`, `names`, and `documentation`. The rascal code is displayed in Rascal 2. The characteristics of the elements are described in the paragraphs below.

Declarations `Declarations` defines the declarations of namespaces, classes, interfaces, 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

Rascal 2 M^3 core definitions in Rascal

```
anno rel[loc name, loc src] M3@declarations; // maps declarations to file location.
anno rel[loc from, loc to] M3@containment; // what is logically contained in what else
anno rel[loc definition, Modifier modifier] M3@modifiers; // associated modifiers
anno rel[loc src, loc name] M3@uses; // maps src locations of usages to the declarations
anno rel[str simpleName, loc qualifiedName] M3@names; // end-user readable names
anno rel[loc definition, loc comments] M3@documentation; // comments and doc-blocks
```

logical name of a class can be |`php+class://SomeNameSpace/ClassX`| while the actual location might be |`file:///project/SomeNameSpace/ClassX.php`|.

Containment `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 highest level) because all functions are declared as first class citizens in PHP.

Modifiers `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.

Uses `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.

Names `Names` contains the declaration and a simplified version of the name. The declared name can be long and unreadable. `names` contains human readable name which can be used for presenting the element in a GUI.

Documentation `Documentation` contains the link to the documentation related to the source code element. The link is mapped to a declaration.

5.1.2 PHP specific elements

Because every programming language differs in syntax and semantics, the M^3 model is extensible to provide language specific elements. The following php specific items are added: `extends`, `implements`, `parameters`, `constructors`, and `annotations`. These PHP specific elements are described in more detail in the paragraphs below.

Rascal 3 PHP specific M^3 element

```
anno rel[loc from, loc to] M3@extends;    // which class extends which class
anno rel[loc from, loc to] M3@implements; // interface usages
anno rel[loc decl, PhpParams params] M3@parameters; // formal functions/methods parameters
anno rel[loc decl, loc to] M3@constructors; // constructor usages
anno rel[loc pos, Annotation annotation] M3@annotations; // annotations from doc blocks
```

Extends `Extends` contains information about what classes and interface extend other classes or 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.

Implements `Implements` holds information on which class implements which interfaces. One class can implement no, one, or 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.

Parameters `Parameters` keeps track of the parameters of a method or function. `PhpParams` is a list relation and contains the optional typehint, if the parameter is required and if it is passed as reference. This information is stored to make it easier to resolve the call to a method or function.

Constructors `Constructors` lists 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.

Annotations `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 regular expressions (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`.

5.1.3 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.

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);
```

The input of the algorithm is all `PHP files of a program`, which are all files ending on `.php` within a program. **The output** is an M^3-model 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 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

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

line 5 can be `namespace`, `class`, `interface`, `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.

5.1.4 Rascal implementation

The constraints are implemented in Rascal with the `TypeOf` and `Constraint` definitions, which are shown in Rascal 4.

The `TypeOf` definition represents a literal type or the type of an expression. To define a literal type, we use the `typeSymbol` which takes a `TypeSymbol` as argument. TypeSymbols are defined in Rascal 1. We use the `typeOf` definition, which takes a Rascal location (loc) as argument, when we define the type of an expression or statement. The location represents the physical location of the expression in the source code.

The `Constraint` definition consists of recursive and non-recursive definitions. `eq` states that the type of the left hand side should be equal to the type of the right hand side. `subtyp` states that the type of the left hand side should be a sub type of the right hand size. This sub type relation is self inclusive, which means that the type can also be the same. `supertyp` is the opposite of `subtyp`, and so the type of the left hand side should be a super type of the right hand size and cannot be the same type. `isMethodOfClass` takes as arguments an two expressions and a string. The two expressions represent the possible types for the left and right hand side of the expression. The string contains the name of the called method.

The recursive definitions allow support for negation, conjunction, disjunction. The `negation` is added to make it easy invert complex constraints, for example when you can say that the type of an expression can be anything but a string. `conjunction` and `disjunction` are used for `and` and `or` relations.

Rascal 4 TypeOf and Constraint definitions

```
data TypeOf
    = typeSymbol(TypeSymbol ts)
    | typeOf(loc ident)
    ;

data Constraint
    = eq(TypeOf lhs, TypeOf rhs)
    | subtyp(TypeOf lhs, TypeOf rhs)
    | supertyp(TypeOf lhs, TypeOf rhs)
    | isMethodOfClass(TypeOf expr, TypeOf classVariable, str name)

    | disjunction(set[Constraint] constraints)
    | conjunction(set[Constraint] constraints)
    | negation(Constraint constraint)
    ;
```

5.2 Constraint extraction

In section 4.1 we've defined the constraint extraction rules. These constraint rules are implemented in Rascal with the help of visiting[2] and pattern matching[3]. Visiting is a concept implemented in the core of Rascal and is based on the visitor design pattern. This makes it very easy to traverse trees like AST's, without coupling the design pattern to your code. During the visiting of the AST we use pattern matching on function arguments to find to correct functions to handle the constraint extraction.

5.2.1 Code snippet

A snippet of the constraint extraction implementation[4] is shown in Rascal 5. In this snippet we show a small subset of the code used in the constraint extraction process.

The code starts with a local declaration of `constraints`, an empty set. This set of constraints will be filled in the rest of the functions on this Rascal module.

You can get all the constraints of a program by calling the only public function `getConstraints` with a `System` and `M3` as parameters. `System` represents a collection of all AST's of a program. The provided `M3` is the M^3-model of the whole program, containing facts about the program. In this function we loop over all `Scripts` of a system and call `addConstraints` for each script.

Using argument pattern matching the function `addConstraints(Script script, M3 m3)` is triggered. The first thing this function executes is `addConstraintsOnAllVarsForScript`, where we add the constraints that the types of a variable should be of one type within a scope. Next we loop over all the statements of the body of a `script`.

Because of pattern matching, the function `private void addConstraints(Stmt statement, M3 m3)` will be called for each statement. In this function we execute a *top-down-break visit*. *Top-down* means that the visit will start at the root node and then moves down towards the leaves. *Break* means that the in dept visit will stop as soon as there is a pattern match. We need to include *break* in this visit because for some nodes we need context. For example if a variable names `$this` would match, we need to know in which class this item was declared. In the code snippet we've only included `exprstmt`, a match with a PHP expression.

The last function of the snippet, `private void addConstraints(Expr e, M3 m3)`, handles expressions like assign with operator, a string scalar, and PHP variables that has a name. With this small code snippet we can show the constraint extraction with a tiny example.

Tiny example With this part of the code, we could extract constraints for the file with the content: `<?php $a += '1';`. When we extract the constraints from this file we would come to the following constraints:

- the type of `$a` is a subtype of `numberType()`
- the type of `$a` is a subtype of `any()`
- the type of `'1'` is equal to `stringType()`

See section 4.1 to see the full list of constraints that we extract from the source. The full implementation can be found on github.

[2] http://tutor.rascal-mpl.org/Rascal/Concepts/Visiting/Visiting.html
[3] http://tutor.rascal-mpl.org/Rascal/Concepts/PatternMatching/PatternMatching.html
[4] The complete source code can be found in https://github.com/ruudvanderweijde/php-analysis

Rascal 5 Constraint extraction snippet

```
private set[Constraint] constraints = {};

public set[Constraint] getConstraints(System system, M3 m3) {
  for (s <- system)
    addConstraints(system[s], m3);
  return constraints;
}

private void addConstraints(Script script, M3 m3) {
  addConstraintsOnAllVarsForScript(script, m3);
  for (stmt <- script.body)
    addConstraints(stmt, m3);
}

private void addConstraints(Stmt statement, M3 m3) {
  top-down-break visit (statement)
    case exprstmt(Expr expr):  addConstraints(expr, m3);
}

private void addConstraints(Expr e, M3 m3) {
  top-down-break visit (e) {
    case a:assignWOp(Expr assignTo, Expr assignExpr, Op operation):  {
      addConstraints(assignTo, m3);
      addConstraints(assignExpr, m3);

      switch(operation)
        case plus():  constraints += {subtyp(typeOf(assignTo@at), numberType())};
    }
    case v:var(name(name(name))):
      constraints += {subtyp(typeOf(v@at), \any())});

    case s:scalar(Scalar scalarVal):
      switch(scalarVal)
        case string(_):  constraints += {eq(typeOf(s@at), stringType())};
  }
}
```

5.3 Constraint solving

When all constraints are extracted from the source code, 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. In this initial phase the types of each object is of type any, except for the literal types which can be resolved already. 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, resulting in an empty set of possible types.

5.3.1 The algorithm

In algorithm 2 we show the algorithm to solve the constraints.

Algorithm 2: Constraint solving algorithm

Input: set[Constraint] constraints
Output: map[TypeOf expression, set[Type] possibleTypes] solutions

```
1 map[TypeOf, set[Type]] solutions = init(constraints);
2 while changes in constraints or solutions do
3     constraints = constraints + deriveMore(constraints, solutions);
4     solutions = propagateSolutions(constraints, solutions);
5 end
6 return solutions;
```

The algorithm input is a set of constraints and the output is a map of solutions. The set of constraints are the constraints defined in constraint extracting section, and the map of results is the location of all type-able objects with their possible solutions. The init function on line 1 adds all type-able objects to the set of possible solutions. The initial possible types of the objects will be the collection of all possible types, called $Universe()$. Only the literal types can be resolved already, like strings, numbers and boolean constants.
Line starts a loop until a fixed point is reached. In Rascal this method is called `Solve()` and will loop until there are no more changes in the given data. The methods of line 3-4 may alter the values of constraints or solutions.
The first function of the loop on line 3, `deiveMore`, visits all constraints and checks if there can be new constraints extracted based on the latest constraints and solutions. Example of these new constraints are conditional constraints, which could only be added if more information of the conditional was available. Function `propagateSolutions` on line 4 propagates resolved estimates. Here we take the intersection of the known solutions with the given constraints.

5.4 Additional constraints

We've added the option to add context to the type inference process. These are the ability to include type annotations from phpdoc in the constraint analysis, and to include constraints on php built-ins.

Annotations After we gather all the facts from the source code, when constructing an M^3 model, we have added type information from annotations. We have tried to write a grammar for the PHPDocs but because their lack of consistency the grammer would fail too many times and would not give us the wanted results, so we went for the regular expression matching instead. We match `@return` and `@param` annotations for functions and class methods, and read `@var` annotations 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.

PHP Built-ins In order to see if we can resolve more types by providing type information of PHP built-ins. The built-in features are represented by PHP files and can also be parsed. Also the annotations of these PHPDocs should be parsed. One special thing that we do is that we not add constraints for the empty body of the method, because this would result in an empty return values which is invalid. We also filter the results type resolutions of the php built-in types out of the results we provide in the next chapter.

Chapter 6

Evaluation

In this chapter we evaluate the type inference algorithm. We first describe the setup of the experiment in section 6.1. Next we present the results in section 6.2. In section 6.3 we analyse the measured results and explain the differences between the performed analysis.

6.1 Experiment setup

In order to validate the this research we tested our type inference algorithm on a selection of the 30 most popular packages of Packagist[1]. Packagist is a repository for Composer[2]projects. Composer is a dependency manager for PHP. All projects have between 2 and 6 million downloads, so they should be suitable for our research because they are frequently used in live applications. The selection of projects, sorted on total lines of code, is listed in table 6.1. The statistics are generated using phploc[3].

Product			Files		Objects			Lines of code		
Vendor	Project*	Version	D[1]	F[2]	C[3]	I[4]	T[5]	Total ↑	Logical	
doctrine	lexer	v1.0	2	7	3	0	0	733	128	(17.46%)
phpunit	php-timer	1.0.5	5	11	5	0	0	740	117	(15.81%)
phpunit	php-text-template	1.2.2	5	11	5	0	0	768	125	(16.28%)
doctrine	inflector	v1.0	2	7	3	0	0	853	130	(15.24%)
psr-fig	log	1.0.0	3	15	8	2	2	1 039	155	(14.92%)
phpunit	php-file-iterator	1.3.4	5	13	7	0	0	1 071	176	(16.43%)

*Selection of the 30 most popular packagist packages ordered by LOC, in July 2014.
[1] = Directories, [2] = Files, [3] = Classes, [4] = Interfaces, [5] = Traits

Table 6.1: Statictics of most populair composer projects

Type inference In this research we are interested in resolving types for expressions. As explained in section 5.3, the result of the constraint solving is a set of types for each expression. We group these resulting typesets in two groups, resolved and unresolved types. Resolved type means that we can deduce the typeset to one type. All other types are unresolved. In unresolved we include typesets that are not resolved and have only type `any()`, empty typesets, and multiple types that cannot be reduced to one unique type for an expression.

Annotations We first run the type inference without reading annotations from PHPDocs. As described in section 4.2, we can use type annotations to read types of variables, functions, methods, and class attributes. Our goal is to see if it is possible to resolve more types when we take the type annotations from the PHPDocs into account.

[1]https://packagist.org/explore/popular, July 2014
[2]https://getcomposer.org/, July 2014
[3]https://github.com/sebastianbergmann/phploc, July 2014

Built-ins After we run the analysis with and without PHPDocs, we run the analysis also with built-in information. Modern editors stub files written in PHP which represent the internal behaviour of constants, variables, functions, classes, methods and class attributes. Using the type information of the built-in language constructs we want to see if we are able to resolve more types.

Expectations The goals of these three analysis is to find out of taking more context into account will result in. We expect that when we take annotations into account that we are able to resolve more types. This is because there is a lack support for types in the language itself. This is mainly caused because it is a scripting language. When we add PHP built-in information to the analysis we expect to resolve slightly more information. We've seen in the research on PHP features usage[HKV13] that many projects use many built-in features. Because there is no type information available by default we expect to be able to resolve more types when adding type annotations of PHP classes, functions, and variables.

6.2 Results

In this section we present the results of the analysis. We start with the results of the analysis with and without taking PHPDocs into account. Next we show the results of the analysis with and without PHP built-ins typehints.

Annotations The results of the analysis without and without taking type annotations from PHPDocs into account are shown in table 6.2. In this table on the left side we present the name of the project and the total number of expressions that were attempted to be inverted. On the right side of the table we present the percentage of resolved types for the analysis with and without taking the PHPDoc context into account.

Project	Total	Resolved types w/o PHPDoc	Resolved types w/ PHPDocs
php-timer	14	3	9
log	66	27	27
php-text-template	23	8	10
lexer	42	19	24
inflector	31	3	14
php-file-iterator	63	19	28

Table 6.2: Type inference results, with and without PHPDocs

Built-ins The results of type inference where we take type information of PHP built-in functions and variables into account are shown in table 6.3. In this table we present the analysed projects and the total amount of expressions we attempted to infer. On the right side of the table we present the percentage of resolved types for the analysis with and without built-in information.

Project	Total	Resolved types w/o Built-ins	Resolved types w/ Built-ins
php-timer	14	9	9
log	66	27	27
php-text-template	23	10	10
lexer	42	24	24
inflector	31	14	14
php-file-iterator	63	28	29

Table 6.3: Results of type usage, with and without PHP built-ins

6.3 Analysis

To get better insight on the difference between the results we list the percentages of resolved expressions next to each other in the bar chart in figure 6.1. The three bars represent the three variants of the analysis. The bars respectively present the results of the analysis where we do not take extra context into account, the results with using annotations from PHPDocs, and the results where we also take

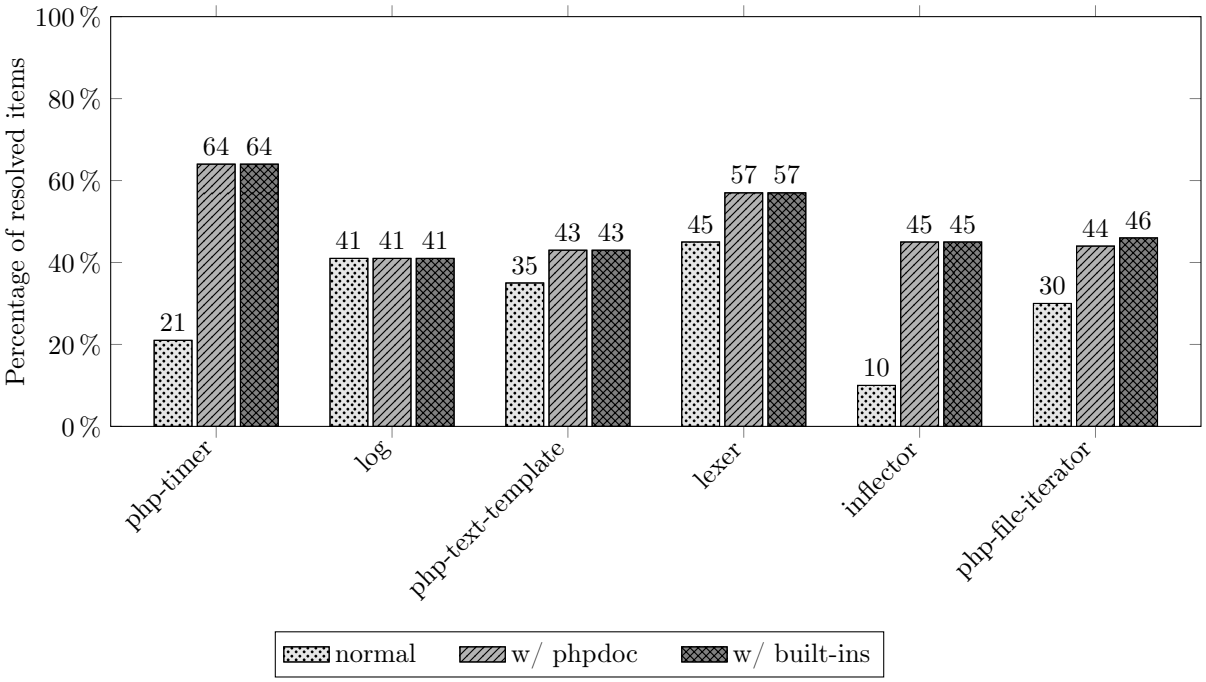


Figure 6.1: Comparison of the results

PHP built-in information into account. The percentages are rounded to the nearest integer to improve readability.

Normal vs PHPDocs In the bar chart we can see that, except for one project, all projects have an increased number of resolved items. There are two projects that show a big difference and three projects that have slightly more resolved items when we take annotations into account. In general we could say that adding annotations aids in having more resolved results.

PHPDocs vs built-ins When we compare the PHPDocs analysis with the built-in analysis we only see a small increase in the number of resolved items for one of the projects. This project is the `php-file-iterator`. All the other projects that we analysed show no difference in the number of resolved item. Because the wide usage of built in functions, classes, and variables we were expecting a bigger difference in the number of resolved items. This might not be the case for these smaller library packages that we used for this analysis. The projects are very small and abstract.

Chapter 7

Conclusion

In this section we present the conclusion in section 7.1

7.1 Conclusion

In this work we have presented a type inference implementation written in Rascal. We presented a theoretical constraint based type inference algorithm. This type constraint algorithm is implemented with a PHP parser written in PHP and constraint extractor written in Rascal. To resolve the constraints we also created a constraint solver in Rascal. The results of the constrain solver is a list of code expressions and their possible set of types.
In a small evaluation we have performed the type inference on a few small projects. Here we found that adding annotations resulted in inferring more types for almost all projects. When we also added type information about PHP built-in classes, functions, and variables, we only saw a small improvement in one of the projects. Hereby we could conclude, although the number of projects is to small to generalise the results, we saw a good improvement when adding type annotations from PHPDocs, but no extra improvement when also adding PHP built-in information.

7.2 Future work

There are various extensions of this research possible. The constraint based type inference could be combined with other analysis, like file include resolutions, dead code elimination, reference/alias analysis, and constant folding/propagation. By including these other analyses it should be possible to get more precise results.
The performance of the constraint solver could be optimised. In our version we didn't take any effort to optimise the algorithm. But we've seen that the algorithm is too slow to be able to run on bigger code bases. Also if you would want to use this real time in an IDE, the run time of the script should be decreased. If the performance of the type inference is improved it is also easier to validate the results of this analysis by adding more projects to the evaluation.
At the moment of writing the implementation was PHP7 not released. The benefit we've seen on adding type annotation is optionally implemented in PHP7. It would be interesting to use the new language constructs and compare the results with PHP5 programs.

7.3 Threats to validity

Because we perform an over approximation of run time values at compile time, the results of the type inference are not 100% accurate. With the dynamic nature of the programming language and the usage of dynamic features it is a challenge to be as accurate as possible.

Missing constraints By adding a constraint we limit the number of possible types for that expression. Because we are humans, it could be possible that we make a mistake by missing a constraint or adding

a wrong constraint. This could result in type inference results that are wrong.

Law of small numbers Because our small sample size in our evaluation, we cannot generalise these results. In order to gain more precise results we should increase our sample size. We can achieve this by running analysis on more software projects in the future.

Object references One thing we do not take into account are the side effects that can be caused by the passing arguments as reference. When an referenced argument is modified, the referenced variable is also modified. This could theoretically lead to type changes.

PHPDoc correctness In this research we also assume that the provided type annotations in the PHPDoc's are correct. These annotations are ignored during program execution and therefore could be incorrect without noticing. Using PHP7 new type hint language constructs could help to ensure that we extract the correct constraints for the source code.

Glossary

AST

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

PSR

PHP Standard Recommendation (PSR) is project which provides rules for commonalities between PHP projects. Autoloading, coding style guide, logging, and HTTP Message interface are the first few accepted standards. The PHPDoc standard describes how and what to use in doc blocks and is currently in draft phase. See http://www.php-fig.org/ for more information. .

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

[Age95] Ole Agesen. "The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism". In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP '95. London, UK, UK: Springer-Verlag, 1995, pp. 2–26. ISBN: 3-540-60160-0. URL: http://dl.acm.org/citation.cfm?id=646153.679533.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[Ayc00] John Aycock. "Aggressive Type Inference". In: *Proceedings of the 8th International Python Conference*. 2000, pp. 11–20.

[Bas+15] Bas Basten et al. "M^3: a General Model for Code Analytics in Rascal". In: *Proceedings of the first International Workshop on Software Analytics, SWAN*. 2015.

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

[Cam07] Patrick Camphuijsen. "Soft typing and analyses on PHP programs". In: (2007).

[CHH09] Patrick Camphuijsen, Jurriaan Hage, and Stefan Holdermans. "Soft Typing PHP with PHP-validator". In: (2009).

[DM82] Luis Damas and Robin Milner. "Principal Type-schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, pp. 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176. URL: http://doi.acm.org.proxy.uba.uva.nl:2048/10.1145/582153.582176.

[Hin69] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". English. In: *Transactions of the American Mathematical Society* 146 (1969), pages. ISSN: 00029947. URL: http://www.jstor.org/stable/1995158.

[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.

[HKV14] Mark Hills, Paul Klint, and Jurgen J. Vinju. "Static, Lightweight Includes Resolution for PHP". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 503–514. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2643017. URL: http://doi.acm.org/10.1145/2642937.2643017.

[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.

[Mil78] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/0022-0000(78)90014-4. URL: http://www.sciencedirect.com/science/article/pii/0022000078900144.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3540654100.

[PS91] Jens Palsberg and Michael I. Schwartzbach. "Object-oriented Type Inference". In: *SIGPLAN Not.* 26.11 (Nov. 1991), pp. 146–161. ISSN: 0362-1340. DOI: 10.1145/118014.117965. URL: http://doi.acm.org.proxy.uba.uva.nl:2048/10.1145/118014.117965.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-oriented type systems*. Wiley professional computing. Wiley, 1994, pp. I–VIII, 1–180. ISBN: 978-0-471-94128-6.

[Shi88] O. Shivers. "Control Flow Analysis in Scheme". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 164–174. ISSN: 0362-1340. DOI: 10.1145/960116.54007. URL: http://doi.acm.org.proxy.uba.uva.nl:2048/10.1145/960116.54007.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. Tech. rep. 1991.

[VH15] Henk Erik Van der Hoek and Jurriaan Hage. "Object-sensitive Type Analysis of PHP". In: *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*. PEPM '15. Mumbai, India: ACM, 2015, pp. 9–20. ISBN: 978-1-4503-3297-2. DOI: 10.1145/2678015.2682535. URL: http://doi.acm.org.proxy.uba.uva.nl:2048/10.1145/2678015.2682535.

[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.