Type inference for PHP

Using annotations to provide more precise results

Ruud van der Weijde

August 19, 2014, 30 pages

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

WERKSPOT
KLUS • KLIK • KLAAR
ruud.van.der.weijde@werkspot.nl
http://www.werkspot.nl

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

Contents

Abstract

PHP is widely used.
Some parts are hard to analyse.
Knowing (object) types can provide better results (no proof for this).
Annotations can (not) help to better resolve types.

Preface

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

Chapter 1

Introduction

1.1 PHP

PHP[1] is a server-side scripting language created by Rasmus Lerdorf in 1995. The original name 'Personal Home Page' changed to 'PHP: Hypertext Preprocessor' in 1998. PHP source files are executed using the PHP Interpreter. The language is dynamically typed, which means that the behaviour of the source code will be examined during run-time. Statically typed languages would apply these modification during compile type. PHP supports duck-typing, which means that the type of an expression can be transformed to another type at a certain point during execution.

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

Popularity According to the Tiobe Index[2] of july 2014, PHP is the 7th most popular programming language. The language has been in the top 10 since its introduction in the Tiobe Index in 2001. More than 80 procent of the websites have a php backend[3]. The majority of these websites use PHP version 5, rather than version 4 or older versions. It is therefor wise to focus on PHP version from 5 and discard the older unsupported versions.

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

1.2 Position

Todo: explain here why it is important to analyse programs: for security analysis and compiler optimisations. The next stuff needs to be rewritten.
As far as we know, there is no constraint based type inference research like this one performed for PHP. That makes this research unique. There have been similar analysis for other dynamic languages, like smalltalk, ruby and javascript.

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

1.3 Contribution

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

- Created an M3 for php.
- Constraint system

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

1.4 Plan

The rest of this thesis is as follows: chapter 2 contains background and related work. Here we will explain important language constructs and explain similar research. In the next chapter 3 the research method is explained, which will explain the steps taken in this the research.

Chapter 2

Background and related work

This chapter explains a few language constructs which are important for this research. Further more Rascal and M^3 will be explained. In the last section of this chapter the related work is described.

2.1 PHP Language Constructs

In this section for this research important language constructs are presented. Explanations of these constructs should help to understand the performed analysis.

Scoping In PHP, all classes and functions are globally available once they are declared. All classes and functions are implicitly public, inner classes are not allowed, and conditional functions (see paragraph about conditional classes and functions) will be available in the global scope. If a class or function is declared inside a namespace, their name will be prefixed with the namespace name. Closures (anonymous functions in PHP) have the same scoping rules as variables, but they can use variables from outside closure by providing them in the use statement. In this research we will not support closures because they are fairly new and not much used in practice.
For variables there are three scopes: global-, function-, and method-scope. There is an exception for some 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 will then be linked to the global variable in the symbol table[1].

Includes In PHP it is possible to include other files during execution. These files will be loaded inline, so if you include another file in the middle of the file the code will be inserted virtually at that place. Todo here: show the results in some extra section. In this research we will assume that all files are loaded. We have checked all projects for duplicate classes and functions but we have found none. Therefor we assume that this does not effect the outcome of the analysis. It is not known if there are effects on inline includes that we do not cover.
According to the coding standard that is used in the php community[2], function- and class-name classes should not appear when using namespaces and autoloading. When a class which is not loaded in memory is instantiated, the autoloading will try to include a file and load the class. The structure of the autoloading is meant to include classes, interfaces, traits and functions and should not have inline code executions which would lead to side-effects.

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

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

try to use the class or function, the script will die with an fatal error (if the class or function was not defined before).

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

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

Listing 2.1: Conditional class and function definitions

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

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

Listing 2.2: Conditional function declaration

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

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

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

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

```
1 class C {}
2 $c = new C();
```

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

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

Listing 2.3: Dynamic class property

Annotations In PHP Annotations are not part of the official language. They are however widely used. For instance in ZEND, Symfony and Doctrine you can write business logic rules in the form of annotations. These annotations will be parsed and used in real code.
Other annotations are placed on top of classes, methods, functions, and variables. These annotations will help the developers to better understand what the code does. For example you can see what kind of input and output is expected for a method. IDE's will also use this information to better analyse the source code.
Writing annotations is not yet in the PSR standards for PHP, but there is a proposal[4]. For this research we will only focus on the `@param`, `@return`, `@var`, and `@inheritDoc` annotations.

2.2 Rascal

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

2.3 M^3

The M^3-model is a model which can hold information of source code[Izm+13]. This model is created to analyse one single JAVA program or compare two or more JAVA systems with each other. The core of the M^3-model contains `containment`, `declarations`, `document`M^3`ation`, `modifiers`, `names`, `types`, `uses`, `messages`.
The `declarations` relation contains class, method, variable- information with their logical name and their real location. The type of the relation are `locations` and represent the logical name of the declaration and will be used in the rest of the M^3. The `containment` relation has information on what declarations are contained in each other. For example a package can contain a class; a class can contain fields and methods or an inner class; a method can contain variables. The `documentation` relation contains all comments from the source code. The `modifiers` relation has information on the modifiers of declarations. Modifiers are abstract, final, public, protected or private. The `names` relation contains a simplified name of the full declarations. The `types` relation has information about the type of the source code elements. The `uses` relation describes what reference is using which object. For instance when a field of a class is used in some expression, the `uses` relation links the field in the expression to the declaration of the field in the class. And lastly, `messages` contains errors, warnings or info statements.

2.4 Related work

Describe these:

- 'The HipHop Compiler for PHP'[Zha+12] (not much information available on their type inference, only source code)
- 'Phantm: PHP analyzer for type mismatch'[KSK10a; KSK10b] (investigate this in more details, their focus is on finding type errors)
- PHPLint [5](uses a different kind of annotations, not the java like phpdocs)

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

- ‘Soft typing and analyses on PHP programs’[], code implementations: https://github.com/henkerik/typing and https://github.com/marcelosousa/soft-typing-PHP5 (created for php4, code for php5, should check this out, might be able to compare results with this)
- ‘Design and Implementation of an Ahead-of-Time Compiler for PHP’[Big10] (to check in detail)

Also describe their differences with my research.

[5] http://www.icosaedro.it/phplint, july 2014

Chapter 3

Research Method

3.1 Research Question

The research question will be something like:

> Will the use of annotations[1] provide more resolution for constraint based type inference?

Subquestions:

- Where do the differences come from?
- Which differences have a positive influence on the results?
- Which differences have a negative influence on the results?
- Can we say something about the reliability of the annotations?

3.2 Context

The following items should be wrapped in a piece of text. (Design decisions??)

- We assume that the program is correct. This means that warnings can happen, but fatal errors not.
- We assume that all files are included.
- We assume that register globals is off! (maybe add some other runtime environments)
- Ingore warnings (because most production code has them off)
- Our analysis is flow-, control-, and context-insensitive (explain what this means)

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

- Analysis is flow insensitive
- Closure
- References
- Variable constructs (variable -variable, -method/function calls, -class instantiation, eval) :: todo: explain WHY not.
- Yields

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

Chapter 4

Research

4.1 Introduction

<< Todo: properly introduction of this chapter >>

4.2 M^3 for PHP

We have constructed a similar M^3 for PHP as was already done for JAVA. Some language constructs are different and are therefor not applicable for PHP and the other way around. For the overlapping constructs we created a similar structure. The model will be used to query the system for information. The following steps are performed to create an M^3 for a system.

- Parse all PHP files, resulting in an AST.
- For each file, create an independent M^3 from the ast.
- Combine all the m3s into one M^3 for the project.
- For each file, add additional informationNow run more analysis when all facts are collected in the m3.
- Done. M3 is finished!

*Maybe add this section to the next chapter, for the research methods. Many aspects will be useful for the type inference analysis
Our goals is to provide the results in an M3 model. Future research can use this to compare different programming languages.
The following core items are filled for M3:

- Containment
- Declarations (need to explain in more details)
- Modifiers
- Extends
- Uses (explain in details what is and what is NOT covered)

The following php specific items are added:

- Extends (class or interface and their extended class or interface)
- Implements (which class implements which interfaces)
- TraitUses (which class uses which trait)
- Parameters (methods and functions and their parameters)
- Constructors (which class uses which constructor, explain this more)
- Aliases (class aliases, for example the usage of class_alias)
- Annotations (contraints annotations on classes, methods, fields and variables)

Todo, explain in more details...
The type inference analysis will be performed in this order:

- M3 stuff...
- Resolve types
- Resolve sub type relations
- Extract facts from the source code
- Extract constraints from source code
- Solve the constraints
- Add result to m3 (all decls)

In the next step, annotations will be added to see how the results be like.

4.3 Put this somewhere

Todo: put this text somewhere....

- We assume that the program is correct. This means that warnings can happen, but fatal errors not.
- We assume that all files are included.
- We assume that register globals is off! (maybe add some other runtime environments)
- Ingore warnings (because most production code has them off)

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

- Analysis is flow insensitive
- Closure
- References
- Variable constructs (variable -variable, -method/function calls, -class instantiation, eval) :: todo: explain WHY not.
- Yields

4.4 Types

Explain here what I mean with a type...

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

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

4.4.1 PHP types

PHP has a similar class inheritance structure and interface implementation as Java. The main difference is that in PHP all class are `public` and that inner classes are not allowed in PHP.
The basis types in PHP are integers, floats (similar to doubles and reals), booleans, strings, arrays, resources and null. When variables are initialised without a values, they are null. The recourse type is a special one which is not important for this research.

4.4.2 Subtypes

Explain something about subtypes here. For now, only this figure4.1

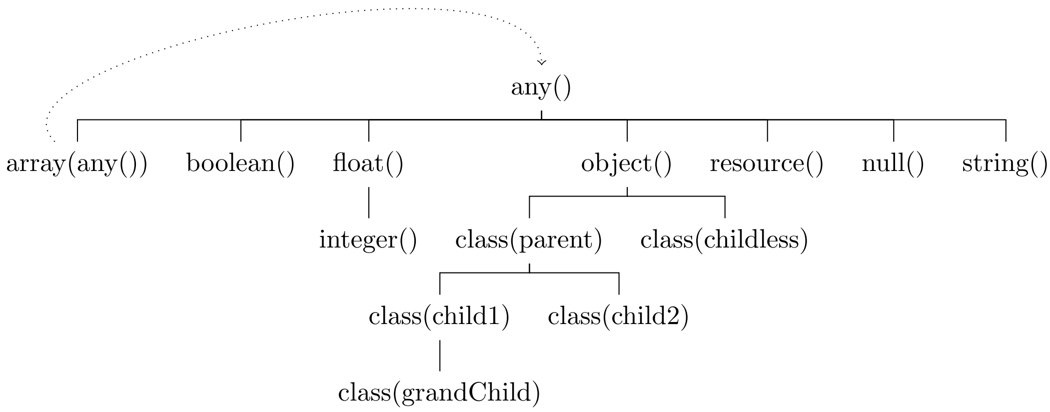
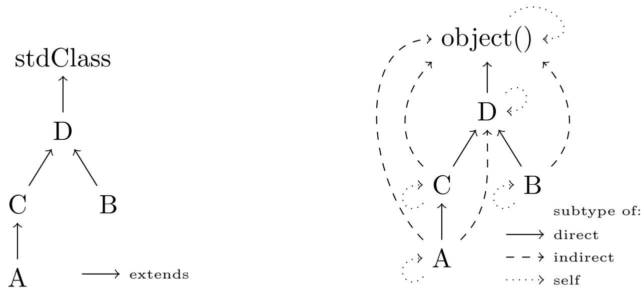


Figure 4.1: Subtype hierarchy

The subtype relation of class inheritance is a reflexive transitive closure relation. A class extension of `class A` on `class C` will define `class A` as a subtype of `class C` in our analysis, as you can see in figure 4.2. If a class does not extend another class, it will implicitly extend the stdClass class. You can see that this happens with `class D` in the example. The `stdClass` is represented as the type `object()` in our analysis.
The basic PHP types also contain a subtype relation. Integers are subtypes of floats.



(a) Inheritance relation

(b) Subtype relation

```
class A extends C {}
class B extends D {}
class C extends D {}
class D {}
```

(c) Inheritance in PHP

Figure 4.2: Relation of subtypes among classes

4.5 Fact extraction

We can extract fact about classes, class-constants/fields/methods, functions, parameters. For these facts, we can use a relation, so we have a many-to-many relation. On the left size we will have the class, function or method. On the right side we have their attribute.
A list of properties: (todo: rewrite this list into a ‘normal’ section.)

- <loc classDecl, className(str name)>
- <loc classDecl, classMethod(str name, set[Modifier] modifiers)>
- <loc classDecl, classProperty(str name, set[Modifier] modifiers)>
- <loc classDecl, classConstant(str name, set[Modifier] modifiers)>
- <loc classDecl, classConstructorParameters(list[PhpParam] params)>
- <loc methodDecl, methodName(str name)>
- <loc methodDecl, methodParameters(list[PhpParam] params)>
- <loc functionDecl, functionName(str name)>
- <loc functionDecl, functionParameters(list[PhpParam] params)>

In Rascal:

```
alias TypeFacts = rel[loc decl, Fact fact];

data Fact
    = className(str name) // = FQN = fully qualified name
    | classMethod(str name)
    | classProperty(str name)
    | classConstant(str name)
    | classConstructorParameters(list[PhpParam] params)
    | methodName(str name)
    | methodParameters(list[PhpParam] params)
    | functionName(str name)
    | functionParameters(list[PhpParam] params)
    ;
```

Other facts that will be used:

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

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

4.5.1 Type extraction

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

```
visit (system) {
    case c:class(_, _, _, _, _):  types += class(c@decl);
}
```

Once all types are defined, we can add the subtype relation. We will need to have the subtype of `int()` and `float()` and the class extensions. You can see that in the code below:

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

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

4.5.2 Constraint extraction

Introduction is needed here... for now I will just list the types that I have found. Maybe this needs to be moved to a different chapter.
This is a list of items which are not supported (yet):

- References (in PHP they are symbol table aliases)
  - on expression assignments :: $\$a = \&\$b$
  - on functions :: function $\&f()$ $\{\ldots\}$
  - on parameters :: function $f(\&\$a)$ $\{\ldots\}$

- Variable structures:
  - ~~Variable variables~~ :: $$$a$;
  - ~~Variable class instantiation~~ :: *new* $$a$;
  - ~~Variable method or function calls~~ :: $$a()$;
- List assign :: $list(\$a, \$b) = array("one", "two")$; (we can assume that the rhs is of type array, when the program is correct)
- ~~Method or function parameters (including type hints)~~
- ~~Class structures, method calls~~
- ~~Class Constants~~
- ~~The global statement~~ (should be resolved by the usage relation from M3)
- ~~Casts of expressions~~
- Parameters
- ~~Predefined variables~~ ($this, self, parent, static)
- ~~Eval~~ (will not be supported)
- ~~Closures~~ (not used much in production code)
- ~~Traits~~ (not used much in production code)
- ~~Callable~~ (introduced in 5.4 as typehint, not used much in production code)
- Foreach($a as ... (=> ...)) => $a is an array or an object;
- ~~return; => return type is null~~ (is added to the situation when there are no return statements)
- add predefined globals (and their type: $[GLOBALS, _SERVER, _GET, _POST, _REQUEST, _COOKIE, _ENV, _SESSION, php_errormsg] (all in global scope))
- add magic constants: __[DIR, FILE, LINE, NAMESPACE, FUNCTION, CLASS, METHOD]__
- ~~predefined constants: TRUE(b), FALSE(b), NAN(f), INF(f), NULL(n), STDIN(r), STDOUT(r), STDERR(r)~~
- define("name", value) mixed with constants (?out of scope?)
- ~~keywords: self, parent, static in a class~~ (is included in method and property calls)
- ADD CONSTANTS! RECORD THE TYPE OF THE DEFINED CONSTANTS AND TRY TO READ THEIR TYPE.

Legend

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

Table 4.1: Constraint legend

A list of predefined items can be found here:

- Constants
- Variables
- todo: functions
- todo: classes

Expressions

Normal assignment

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

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

Listing 4.1: Normal assignment

Ternary

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

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

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

Listing 4.2: Ternary

Assignments with operators (1) always resulting in ints

$$\frac{\begin{array}{c}E_1\ \&= E_2\\ E_1\ |= E_2\\ E_1\ \hat{}= E_2\\ E_1\ <<= E_2\\ E_1\ >>= E_2\\ E_1\ \%= E_2\end{array}}{[E_1] = integer()}$$

```
$a &= $b;  /* $a = integer() */
$a |= $b;  /* $a = integer() */
$a ^= $b;  /* $a = integer() */
$a <<= $b; /* $a = integer() */
$a >>= $b; /* $a = integer() */
$a %= $b;  /* $a = integer() */
```

Listing 4.3: Assignments with operators (1)

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

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

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

Listing 4.4: Assignments with operators (2)

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

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

```
$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 4.5: Assignments with operators (3)

Assignments with operators (4) resulting in int or float

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

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

Listing 4.6: Assignments with operators (4)

Unary operators

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

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

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

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

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

```
+$a // positive
-$a // negation
!$a // not
~$a // bitwise not
$a++ // post increase
$a-- // post decrease
++$a // pre increase
--$a // pre decrease
```

Listing 4.7: Unary operators

Binary operators

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

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

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

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

```
$a + $b // addition
$a - $b // subtraction
$a * $b // mulitiplication
$a / $b // division
$a % $b // modulus
$a & $b // bitwise And
$a | $b // bitwise Or
$a ^ $b // bitwise Xor
$a << $b // bitwise shift left
$a >> $b // bitwise shift right
```

Listing 4.8: Binary operators

Comparison operators

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

```
$a == $b   /* boolean() */
$a === $b  /* boolean() */
$a != $b   /* boolean() */
$a <> $b   /* boolean() */
$a !== $b  /* boolean() */
$a < $b    /* boolean() */
$a > $b    /* boolean() */
$a <= $b   /* boolean() */
$a >= $b   /* boolean() */
```

Listing 4.9: Comparison operators

Logical operators

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

```
$a and $b /* boolean() */
$a or $b  /* boolean() */
$a xor $b /* boolean() */
$a && $b  /* boolean() */
$a || $b  /* boolean() */
```

Listing 4.10: Logical operators

Array

Array value fetch

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

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

Listing 4.11: Array value fetch

Array declaration

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

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

Listing 4.12: Array declaration

Scalars

Scalars

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

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

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

```
"Str" // string()
'abc' // string()
100 // integer()
1.4 // float()
```

Listing 4.13: Scalars

Encapsulated strings

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

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

```
"$var"
```

Listing 4.14: Encapsulated strings

Casts

Casts
Note: PHP Warnings are ignored

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

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

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

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

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

$$\frac{E \equiv (string)E_1}{[E] = string(),}$$

$$if([E_1] <: object()) => hasMethod([E_1], `\_\_tostring')$$

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

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

Listing 4.15: Casts

Clone

Clone

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

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

Listing 4.16: Clone

Class

Class instantiation (1) matching the class name

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

```
new C;
```

Listing 4.17: Class instantiation (1)

Class instantiation (2) of an expression

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

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

Listing 4.18: Class instantiation (2)

Special keywords `self` `parent` `parent` `static`

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

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

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

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

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

Listing 4.19: Special keywords

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

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

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

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

$$\frac{E_1 \rightarrow E_2 \subseteq C_1*}{\begin{array}{c}[E_1] = C_1.\text{hasProperty}(E_2.\text{name, static} \notin \text{Mfs}) \vee \\ [E_1] = C_1.\text{parent.hasProperty}(E_2.\text{name, public|protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee \\ [E_1] = C.\text{hasProperty}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})\end{array}}$$

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

$$\frac{E_1 \rightarrow E_2 \nsubseteq C \subseteq \Gamma*}{[E_1] = \text{C.hasProperty}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})}$$

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

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

Listing 4.20: Class property fetch

Class property fetch variable

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

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

Listing 4.21: Class property fetch variable

Class method call

$$\frac{E \equiv \$this \rightarrow E_1 \subseteq C_1}{\begin{array}{c}[\$this] <: \text{object}(),\ [\$this] = \text{class(C)} \vee [E] :> \text{class(C)}, \\ [E_1] \text{ isMethod}(),\ [E_1] \text{ hasName}(E_1.name \vee \text{"\_\_call"}), \\ [E_1] = C_1.\text{hasMethod}(E_1.\text{name, static} \notin \text{Mfs}) \vee \\ [E] =?\end{array}}$$

$$\frac{E \equiv self :: E_1 \subseteq C_1}{\begin{array}{c}[E_1] = C_1.\text{hasMethod}(E_1.\text{name, static} \in \text{Mfs}) \vee \\ [E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name, public|protected} \in \text{Mfs} \wedge \text{static} \in \text{Mfs}) \vee \\ [E_1] = C_1.\text{hasMethod}(\text{"\_\_callStatic"})\end{array}}$$

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

$$\frac{E \equiv E_1 \rightarrow E_2 \subseteq C_1*}{\begin{array}{c}[E_1] = C_1.\text{hasMethod}(E_2.\text{name, static} \notin \text{Mfs}) \vee \\ [E_1] = C_1.\text{parent.hasMethod}(E_2.\text{name, public|protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee \\ [E_1] = C.\text{hasMethod}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})\end{array}}$$

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

$$\frac{E \equiv E_1 \rightarrow E_2 \nsubseteq C \subseteq \Gamma*}{[E_1] = \text{C.hasMethod}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})}$$

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

This stuff is old!!!!!!

$$\frac{\$this \rightarrow E_1 \subseteq C_1}{\begin{array}{c}[E_1] = C_1.\text{hasMethod}(E_1.\text{name, static} \notin \text{Mfs}) \vee \\ [E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name, public}|\text{protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee \\ [E_1] = C_1[\text{.parent}].\text{hasMethod}(\texttt{"\_\_call"})\end{array}}$$

$$\frac{self :: E_1 \subseteq C_1}{\begin{array}{c}[E_1] = C_1.\text{hasMethod}(E_1.\text{name, static} \in \text{Mfs}) \vee \\ [E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name, public}|\text{protected} \in \text{Mfs} \wedge \text{static} \in \text{Mfs}) \vee \\ [E_1] = C_1.\text{hasMethod}(\texttt{"\_\_callStatic"})\end{array}}$$

$$\frac{parent :: E_1 \subseteq C_1}{\begin{array}{c}[E_1] = C_1.\text{parent.hasMethod}(E_1.\text{name, public}|\text{protected} \in \text{Mfs}) \vee \\ [E_1] = C_1.\text{parent.hasMethod}(\texttt{"\_\_callStatic"})\end{array}}$$

$$\frac{E_1 \rightarrow E_2 \subseteq C_1*}{\begin{array}{c}[E_1] = C_1.\text{hasMethod}(E_2.\text{name, static} \notin \text{Mfs}) \vee \\ [E_1] = C_1.\text{parent.hasMethod}(E_2.\text{name, public}|\text{protected} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs}) \vee \\ [E_1] = C.\text{hasMethod}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})\end{array}}$$

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

$$\frac{E_1 \rightarrow E_2 \nsubseteq C \subseteq \Gamma*}{[E_1] = \text{C.hasMethod}(E_2.\text{name, public} \in \text{Mfs} \wedge \text{static} \notin \text{Mfs})}$$

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

```
$this->methodCall();
self::methodCall();
parent::methodCall();
$a->methodCall();
$a::methodCall();
```

Listing 4.22: Class method call

Class method call variable

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

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

Listing 4.23: Class method call variable

Class constants (needs to be reviewed)

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

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

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

```
self::CONST
parent::CONST
SOMECLASS::CONST
```

Listing 4.24: Class constants (needs to be reviewed)

Parameters

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

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

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

Listing 4.25: Parameters in class instantiation

Scope

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

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

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

Listing 4.26: Type of a certain variable within some scope

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

$$\frac{\text{return} \not\subseteq f \vee \text{return;} \subseteq f}{[f] = null()}$$

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

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

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

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

```
function f() {
  if (rand(0,1))
    return $a;
  else
    return $b;
}
// returns typeOf($a) or typeOf($b)
```

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

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

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

```php
function f() {
  if (rand(0,1))
    return $a;
  else if (rand(0,1))
    return $b;
}
// returns typeOf($a) or typeOf($b) or null()
```

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

Function calls

Function call

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

```php
function f() {}
f();
```

Listing 4.30: Function call

Function call variable

$$\frac{E \equiv E_1() \subseteq \Gamma}{[E] = \mathit{any}(),}$$
$$[E_1] <: \mathit{object}() \vee [E_1] = \mathit{string}(),$$
$$if([E_1] <: \mathit{object}()) => \mathit{hasMethod}(''\_\_invoke'')$$

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

Listing 4.31: Function call variable

How to resolve expressions:

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

4.6 Annotations

Explain how the annotations are added to the constraints.

4.7 Constraint solving

Explain how we will solve constraints.

Chapter 5

Results

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

5.1 Annotations

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

Chapter 6

Case Study

Explain how the case study is performed.

Chapter 7

Conclusion

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

7.1 Conclusion

7.2 Future work

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

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

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

Glossary

AST

Abstract Syntax Tree, explain... .

Rascal

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

reflexive transitive closure

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

stdClass

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

Bibliography

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

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

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

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

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

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

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