Type inference for PHP

Using annotations to provide more precise results

Ruud van der Weijde

August 5, 2014, 27 pages

Supervisor: Jurgen Vinju
Host organisation: Werkspot, http://werkspot.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

T.B.A

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.

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 including improved the OO support. To be able to resolve conflicts between library and create better readable class names, namespaces were added to the release of PHP 5.3 in 2009. 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.

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 version 3. It is therefor useful to focus on PHP version from 5 and disgard the older unsupported versions.

Although the popularity for more than a decade, there is still a lack of good PHP code analysis tools. These tools can help to reveal security vulnerabilities or bugs in source code. The tools can also provide code completions to developers or make automatic transformations on the code possible, for example to execute refactoring patterns. Other dynamic languages suffer the same difficulties

1.2 Position

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. 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 the thesis is as follows: chapter 2 contains background and context information about related work. Here we will explain the php language 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 context

2.1 PHP Language Constructs

In this section important (for this research important!) language constructs are presented. Explanations of these constructs should help to understand the performed analysis. Including some concepts like scope, includes, dynamic variables, dynamic class instantiation, dynamic function call, dynamic dispatch, runtime environment variables and constants, late static binding (static keyword), magic methods.

2.1.1 Scoping

PHP has a few scopes. "Define what scope is". Global, namespace, class, method, function. The global scope is contained in every file which is not inside a function or class. The global scope can contain namespaces. Namespaces are comparable to packages in Java. When namespaces are used, classes and functions will be scoped to the namespace. You can access them by providing the namespace name.
Todo: say something about the global statement and $GLOBALS. (it is resolved in the M3 relation @uses). Refer to this link[1].

- In general:
- Functions are declared in

2.1.2 Includes

Note to myself: how will I deal with this concept in my analysis (totally ignore it??? when maybe not add it to this background information. In our research we will assume that all files are included.). The problem of including files can be reduced 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. For this analysis we will include all files
* Refer to the analysis of mark hills, that most files can be resolved, but not all. We consider the use of including scripts for logic as bad practice. Every file should contain a class, and in this case, it is for our analysis not very interesting to resolve the includes.

2.1.3 Conditional functions and classes

Explain the code below.

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

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

Listing 2.1: Conditional class and function definitions

[1]http://php.net/manual/en/language.variables.scope.php, July 2014

Explain the code below.

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

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

Listing 2.2: Conditional function declaration

2.1.4 Dynamic features

These include dynamic variables, dynamic class instantiations, dynamic function calls.

2.1.5 Late static binding

Late static binding[2]is implemented in PHP since version 5.3 by adding the keyword ‘static’ to the language. It has the same function as ‘parent’ and ‘self’, because they both point to a class. The main difference is that ‘parent’ and ‘self’ can be resolved statically. ‘static’ can only be resolved on runtime and represents the exact class that is instantiated.

2.1.6 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 ‘_ _ construct’.

2.1.7 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.

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

Listing 2.3: Dynamic class property

2.1.8 Annotations

Explain here how annotations work in php. (in the next section I will explain something about parsing/reading annotations for the analysis)

2.1.9 Other concepts

- We assume that register globals is off! (maybe add some other runtime environments)
- Ingore warnings (because most production code has them off)

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

2.2 Rascal

Explain what Rascal is and what we use it for...

2.3 M3

*Maybe add this section to the next chapter, for the research methods. Many aspects will be useful for the type inference analysis
The M3-model is a generic model which can be used to analyse software programs. 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

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...

2.4 Related work

Describe these:

- 'The HipHop Compiler for PHP'[Zha+12] (not much information available, only source code)
- 'Phantm: PHP analyzer for type mismatch'[KSK10a; KSK10b] (investigate this in more details)
- PHPLint [3](uses a different kind of annotations, not the java like phpdocs)
- 'Soft typing and analyses on PHP programs'[], code implementations: https://github.com/henkerik/typing and https://github.com/marcelosousa/soft-typing-PHP5 (created for php4, code for php5, should check this out, might be able to compare results with this)
- 'Design and Implementation of an Ahead-of-Time Compiler for PHP'[Big10] (to check in detail)

Also describe their differences with my research.

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

Chapter 3

Research Method

3.1 Introduction

<< Todo: properly introduction of this chapter >>
The type inference analysis will be performed in this order:

- Resolve types
- Resolve sub type relations
- Extract facts from the source code
- Extract constraints from source code
- Solve the constraints

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

3.2 Research Question

The research question will be something like:

> Will the use of phpdoc annotations produce better (to be defined) results for constraint based type inference?
>
> quotation

3.3 Types

Explain here what I mean with a type...

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

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

3.3.1 PHP types

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

3.3.2 Subtypes

Explain something about subtypes here. For now, only this figure3.1

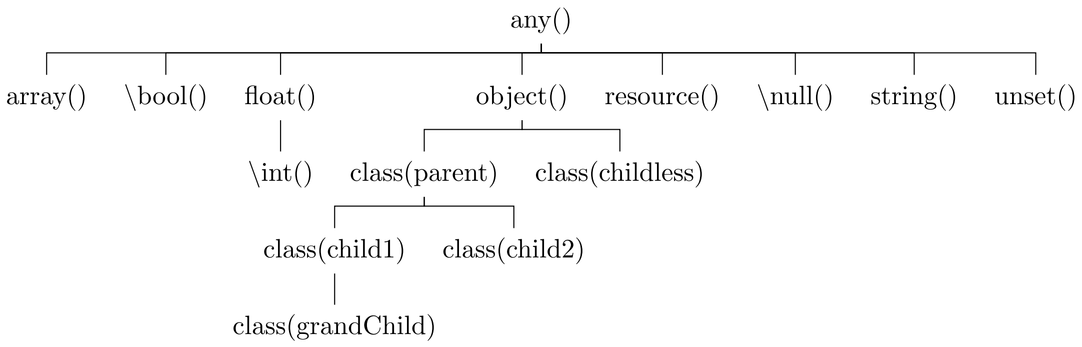
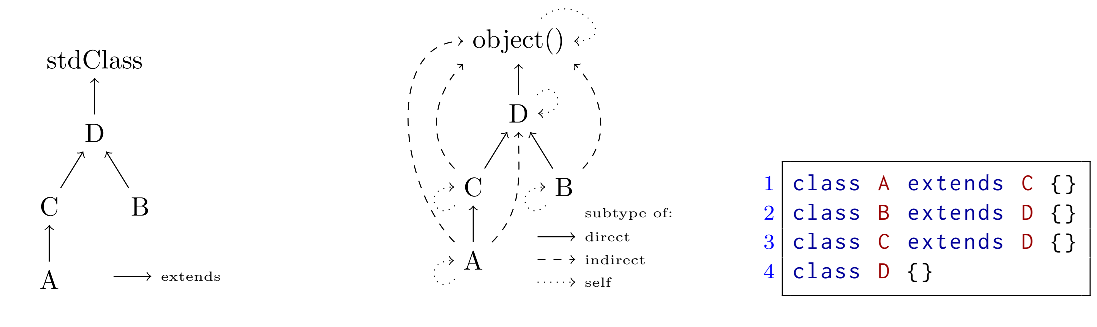


Figure 3.1: Subtype hierarchy

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



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

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

Figure 3.2: Relation of subtypes among classes

3.4 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
```

3.4.1 Type extraction

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

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

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

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

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

3.4.2 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 $\&f()$ $\{\dots\}$
  - on parameters :: function $f(\&\$a)$ $\{\dots\}$
- 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~~

- ~~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

---

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 3.1: Constraint legend

Expressions

Normal assignment

$$\frac{E_1 = E_2}{[E_2] <: [E_1]}$$

```
$a = $b; // normal assign
```

Listing 3.1: Normal assignment

Ternary

$$\frac{E_1 \ ?\ E_2 : E_3}{[E_1 \ ?\ E_2 : E_3] <: [E_2] \vee [E_3]}$$

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

Listing 3.2: Ternary

Assignments with operators (1) always resulting in ints

$$\frac{E_1 \ \&= E_2 \quad E_1 \ |= E_2 \quad E_1 \ \hat{}= E_2 \quad E_1 <<= E_2 \quad E_1 >>= E_2 \quad E_1 \ \%= E_2}{[E_1] = int()}$$

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

Listing 3.3: Assignments with operators (1)

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

$$\frac{E_1 \ .= E_2}{[E_1] = string()}$$

```
$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 3.4: Assignments with operators (2)

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

$$\frac{E_1 \ /= E_2 \quad E_1 -= E_2}{[E_1] = int()}$$

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

Listing 3.5: Assignments with operators (3)

Assignments with operators (4) resulting in int or float

$$\frac{E_1 \ {*}= E_2 \quad E_1 \mathrel{+}= E_2}{[E_1] <: float()}$$

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

Listing 3.6: Assignments with operators (4)

Comparison operators

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

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

Listing 3.7: Comparison operators

3.4.3 Array

Array value fetch

$$\frac{E_1[E_2] \wedge [E_1] == \mathit{string}()}{[E_1[E_2]] = \mathit{string}()}$$

$$\frac{E_1[E_2] \wedge [E_1] == \mathit{array}()}{[E_1[E_2]] <: \mathit{any}()}$$

$$\frac{E_1[E_2] \wedge (E_1 == \mathit{string}() \vee [E_1] == \mathit{array}())}{[E_1[E_2]] = \mathit{null}()}$$

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

Listing 3.8: Array value fetch

Array declaration

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

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

Listing 3.9: Array declaration

Scalars

Scalars

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

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

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

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

Listing 3.10: 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 3.11: Encapsulated strings

Casts

Casts

Note: PHP Warnings are ignored

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

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

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

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

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

$$\frac{(string)E_1}{[(cast)E_1] = string()}$$

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

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

Listing 3.12: Casts

Clone

Clone

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

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

Listing 3.13: Clone

Class

Class instantiation (1) matching the class name

$$\frac{new\ C_1() \in \Gamma,\ \text{C.hasName}(C_1.name)}{[new\ C_1] = \text{class(C.decl)}}$$

```
new C;
```

Listing 3.14: Class instantiation (1)

Class instantiation (2) of an expression

$$\frac{new\ E_1 \in \Gamma}{[new\ E_1] <: \text{object()}}$$

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

Listing 3.15: Class instantiation (2)

Special keywords `self parent parent static`

$$\frac{\$this \in C}{[\$this] = \text{class(C)}}$$

$$\frac{self \in C}{[self] = \text{class(C)}}$$

$$\frac{parent \in C}{[parent] <: \text{class}(C.p^{+[1]})}$$

$$\frac{static \in C}{[static] <: \text{class}(C^{+[2]})}$$

```php
// $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 3.16: 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.

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

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

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

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

*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.

```php
$this->prop // name = prop, vis = public|protected , !static || mm
self::$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 3.17: Class property fetch

[1]all direct or indirect parents
[2]all direct or indirect parents, including the class itself

Class property fetch variable

$$\frac{E_1 \rightarrow E_2,\ E_2 \text{ is an expression}}{[E_1] = \text{object}()}$$

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

Listing 3.18: Class property fetch variable

Class method call

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

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

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

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

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

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

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

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

Listing 3.19: Class method call

Class method call variable

$$\frac{E_1 \rightarrow E_2(),\ E_2 \text{ is an expression}}{[E_1] = \text{object}()}$$

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

Listing 3.20: Class method call variable

Class constants (needs to be reviewed)

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

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

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

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

Listing 3.21: 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))

$$\frac{\begin{array}{cc} \mathit{new}\ C_1\ (A_1,\ A_2,\ \ldots,\ A_k) \subseteq \Gamma & \mathit{class}\ C\ (th_1\ P_1,\ P_2 = v,\ \ldots,\ P_k) \subseteq \Gamma \\ \$a \rightarrow m()\ (A_1,\ A_2,\ \ldots,\ A_k) \subseteq \Gamma & \text{public function } m()\ (th_1\ P_1,\ P_2 = v,\ \ldots,\ P_k) \subseteq \Gamma \\ \mathit{function}_1\ (A_1,\ A_2,\ \ldots,\ A_k) \subseteq \Gamma & \text{function } (th_1\ P_1,\ P_2 = v,\ \ldots,\ P_k) \subseteq \Gamma \end{array}}{[P_1] <: [A_1],\ [A_1] <: [th_1],\ [P_1] <: [th_1],\ \text{hasRequiredParam}(P_1),\ \text{hasRequiredParam}(P_k)}$$

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

Listing 3.22: Parameters in class instantiation

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

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

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

Listing 3.23: Type of a certain variable within some scope

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

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

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

Listing 3.24: Return type of function or method (1)

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

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

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

Listing 3.25: Return type of function or method (2)

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

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

```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 3.26: Return type of function or method (3)

Function call

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

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

Listing 3.27: Function call

Function call variable

$$\frac{E_1() \subseteq \Gamma}{[E_1()] = \mathit{mixed}()}$$

```php
function f() {}
$f = "f";
$f(); // unknown what function will be called
```

Listing 3.28: 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)

3.5 Annotations

Explain how the annotations are added to the constraints.

3.6 Constraint solving

Explain how we will solve constraints.

Chapter 4

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.

4.1 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 5

Case Study

Explain how the case study is performed.

Chapter 6

Conclusion

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

6.1 Conclusion

6.2 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

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).

[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.

[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.