
TAPAs: Type Analysis for PHP Arrays

Christian Budde Christensen, 20103616

Randi Katrine Hillerøe, 20103073

Master's Thesis, Computer Science

June 2015

Advisor: Anders Møller



AARHUS
UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Abstract

► in English... ◄

Resumé

► in Danish . . . ◄

Acknowledgements



...
Aarhus, June 4, 2015.

Contents

Abstract	iii
Resumé	v
Acknowledgments	vii
1 Introduction	1
1.1 Problem Statement	2
1.2 Motivation	2
1.3 Structure of this thesis	3
2 Background	5
2.1 Program structure	5
2.2 Language features	6
2.2.1 Arrays in PHP	6
2.2.2 Scoping	6
2.2.3 References	7
3 Dynamic Analysis	11
3.1 Hypothesis	12
3.2 Implementation	13
3.2.1 Logging of feature usage	14
3.2.2 Identification of arrays	15
3.2.3 Determining type	16
3.2.4 Compiling and running PHP with logging	17
3.2.5 Limitations	17
3.3 Results	18
3.4 Conclusion	21
4 Data Flow Analysis	23
4.1 PHP Language subset	23
4.2 Control flow graph	24
4.2.1 Statements	27
4.2.2 Expressions	28
4.2.3 Reference and variable expressions	31
4.3 Lattice	32
4.4 Transfer functions	36

4.4.1	Operations	36
4.4.2	Variables	37
4.4.3	Arrays	39
4.4.4	Function calls	47
4.4.5	Other transfer functions	49
4.5	Coercion	50
4.6	Abstract evaluation	51
4.7	The Monotone Framework	55
4.8	Implementation	56
4.8.1	Library functions	57
5	Case Study	61
5.1	Maps and lists	61
5.2	Array Pivot	62
5.3	Directory Content	64
5.4	Date Validation	64
5.5	Keeping track of instances	65
5.6	Joining array values	67
5.7	Conclusion	67
6	Related work	69
6.1	Dynamic features	69
6.2	Static Analysis	69
6.3	PHP References	69
6.4	Static Approximation of Dynamically Generated Web Pages . . .	70
6.5	Static Detection of Cross-Site Scripting Vulnerabilities	70
6.6	Static Detection of Security Vulnerabilities in Scripting Languages	70
6.7	Finding Bugs in Web Applications Using Dynamic Test Genera- tion and Explicit-State Model Checking	70
6.8	Sound and Precise Analysis of Web Applications for Injection Vulnerabilities	70
6.9	Alias Analysis for Object-Oriented Programs	70
6.10	Two Approaches to Interprocedural Data Flow Analysis	70
6.11	Practical Blended Taint Analysis for JavaScript	71
6.12	Blended Analysis for Performance Understanding of Framework- based Applications	71
7	Conclusion	73
8	Future Work	75
A	Basic Definitions	77
B	Abstract Operators	79
	Bibliography	83
	Secondary Bibliography	83

Chapter 1

Introduction

PHP is one of the most popular languages for server-side web-development. It is powering over 80% of the web¹, including major websites, such as Wikipedia and Facebook, and the most used CMS's: Wordpress, Joomla, Drupal, and Magento. It requires no compilation and is dynamically typed, which makes development and deployment easy.

As other dynamically typed languages, static type reasoning is non-trivial. This task is only worsened by allowing variable-variables, variable-functions and the all-purpose array datastructure. PHP supports associative arrays with integer and/or string-typed indices (referred to as keys) mapping to values of arbitrary type, including arrays. Combined with the dynamic type system, extensive coercion, and optional error-reporting, debugging becomes difficult and time consuming. Furthermore, there exists no official language specification and the language is thus defined by the reference Zend Engine interpreter.

This thesis will focus on a static type analysis of arrays, which in many existing tools are treated as a black hole where all information is lost. **►maybe cite some related work here?◄** Reasoning about the structure of an array with a decent level of precision seems like an impossible task, since practically no structure is imposed on arrays. But is the imagination of the average PHP developer in practice limited? Are arrays used as other data-structures, such as maps and lists, and can these structures be identified statically? If this is the case, then these structures might be the key to an abstraction yielding a fair compromise between speed and precision for a static type analysis.

By analysing a corpus of existing frameworks dynamically, this thesis aims to identify use-patterns of arrays as other, more restrictive, data-structures. The results of the dynamic analysis is going to motivate the abstraction in a static interprocedural data-flow type analysis on a subset of the PHP language. The static analysis should facilitate error detection by identifying suspicious code.

As mentioned before, error-reporting is optional and throwing a warning or a notice might not stop the program from running. Program 1.1 below will result in a notice being thrown at line 2. The interpreter will assume that `test` should be interpreted as the string `'test'`, following no constant `test` being

¹<http://w3techs.com/>

defined. The program is thus valid PHP, but the intent of the faux string isn't clear, which makes the program suspicious to say the least.

Program 1.1 Suspicious program

```
1 $a = [ 'test' => 42];  
2 echo $a[test];
```

In the following real-life example from the Part framework (a CMS written by one of the authors), the **\$keyArray** and **\$valueArray** arrays are first used as a map in lines 4 and 5, while later, at line 9 and 10 being used as a list, with the *array append* operation. The intentions of this kind of usage is unclear and not very maintainable, but since it ultimately results in the correct behavior, and yields no errors, it is not discovered. The analysis should facilitate discovery of such suspicious cases.

Program 1.2 Mixing array types

```
1 private function createInstance($string, $instance,   
    callable $callback)  
2 {  
3     if (!isset($this->keyArray[$string])) {  
4         $this->keyArray[$string] = [];  
5         $this->valueArray[$string] = [];  
6     } else if (($k = array_search($instance, $this->  
keyArray, true)) !== false) {  
7         return $this->valueArray[$k];  
8     }  
9     $this->keyArray[] = $instance;  
10    return $this->valueArray[] = $callback();  
11 }
```

1.1 Problem Statement

The widely used array data structure in PHP has very few restrictions making it difficult to reason about with program analysis. This thesis will identify possible use-patterns for arrays in PHP and how to detect them in static analysis. An *interprocedural data-flow type-analysis* is proposed to detect suspicious cross-use of the identified patterns.

1.2 Motivation

Creating code that seemingly works is one thing and for languages like PHP that can be done in many different ways. However creating code that conveys a clear purpose and is easily maintainable later on is a whole other world. The latter is preferable most of the time. Because the range of possible uses of PHP arrays is very large they are not conveying a clear purpose in and of themselves.

Creators of PHP programs thus has to ensure themselves the clarity of their use of arrays.

We hypothesize that arrays keep to a specific use-pattern during its lifetime. If the hypothesis holds, a statical analysis can be employed to detect and thereby prevent cross-use of the patterns which in turn will lead to a more clear intention of the code and in the end higher maintainability of the program code.

►add a list of bad practices that hinders clear intention and lowers maintainability◄

1.3 Structure of this thesis

Chapter 2 provides the necessary background knowledge of the PHP language and modern PHP program structure to understand why arrays are difficult to apply existing methods to. In chapter 3 a dynamic analysis is conducted on a corpus of real PHP applications to identify use-patterns and test the hypothesis about use-patterns of arrays. In chapter 4 use-patterns and knowledge gained in the previous chapter is utilized to define and implement a static analysis of a subset of PHP, focusing on detecting suspicious use of arrays. The static analysis implementation is evaluated in chapter 5 by studying interesting cases found in or inspired by the corpus used in the dynamic analysis. Related work is discussed in chapter 6. The last two chapters, 7 and 8, conclude our thesis and describe possible future work.

Chapter 2

Background

PHP (PHP: Hypertext Preprocessor) was created by Rasmus Lerdorf in 1995. As one of the first dedicated web development server-side languages it has become widely popular over the past 20 years. A survey[?] shows that 82% of websites use PHP, including some major websites, such as Facebook, which consequently has created their own gradual typed dialect of the language, called Hack, and HHVM (Hip Hop Virtual Machine), in order to deal with scalability and a enormous code base. The apparent simplicity, availability of the language, and cheap hosting solutions, also enables creation of small websites relatively fast requiring no knowledge of types, concurrency, or objects from the programmer. It is no wonder why PHP, as it is the case for many other programmers, was the first programming language learned by the authors of this thesis.

2.1 Program structure

With the introduction of an object model with PHP 5 in 2005 followed a trend to organize PHP programs in an object oriented manner.

For better organization of code, in 2009, namespaces were introduced (PHP 5.3) and the PHP Framework Interoperability Group, which aims to formulate standards on autoloading, coding style, logging, etc. was created. ►ref◄. Utilizing these tools and standards, many modern frameworks are managing dependencies to other frameworks and libraries through package managers such as Composer ►ref◄. Normally these dependencies are hosted open-source on GitHub, made available for composer on packagist. Composer also handles autoloading of classes, which in practice removes the task of manually loading PHP files using the `include "file.php";` or `require "file.php";` statement.

These statements allows dynamically loading of dependencies, by taking an expression as argument, and resolving these statically has been subject for some research ►ref◄. With autoloading and a proper program structure (as advocated by PHP-FIG) resolving dependencies statically becomes trivial.

With increasing complexity follows a need for quality assurance and testing. PHPUnit is a framework, written in PHP and available via. composer, that facilitates unit-testing. This framework is widely used and e.g. supports testing

with databases and browser output through selenium integration. PHPUnit is also used for generating coverage information for code-analysers, such as code climate, and is natively supported, as a metric of quality, by continuous integration tools, such as travis CI. ►[ref](#)◄.

Travis CI is an online service, free for open-source projects, just as GitHub and Code Climate. It observes GitHub repositories and runs an advanced check on every new committed version, emulating multiple different deployment environments wrt. operating system, PHP interpreter version, web-server, etc. Emulating different environments locally can be done with vagrant which lowers development environment setup time by automating the process, making it possible for developers develop to a production environment.

2.2 Language features

PHP is a ever-changing language, with a new syntax and language features being introduced in every version. Features such as Objects, introduced and modified throughout PHP 5, anonymous functions, introduced in PHP 5.3, variadic functions, introduced in PHP 5.6, etc. Furthermore PHP supports a large number of alias features, i.e. features that semantically are the same, but has different syntax, e.g. array initialization can be written as `array(...)` or `[...]`, array access can be denoted with brackets, `$a['key']`, or curly-brackets `$a{'key'}`, etc. In the rest of this section follows the description of some of the language features that has proven interesting when designing our analysis.

2.2.1 Arrays in PHP

The language construct `array` in PHP is an ordered map. Since *keys* of these ordered maps can be either integer, string or a combination and *values* can be any combination of types, it is possible to use arrays as many different collection types e.g. maps, lists, queues, stacks, trees, dictionaries and probably any other collection-like type that exists.

Arrays can be multidimensional, by containing other arrays, or circular, by containing references to itself. They need not be initialized explicitly, but can be created by performing an array write, e.g. `$a['foo'] = 42;`, or append, e.g. `$a[] = 42;`, on an uninitialized variable or a variable containing `null`, which is practically the same.

2.2.2 Scoping

The variable scoping of PHP is rather simple; there is a unique global variable scope, which is the scope of the statements not part of any function body, a static variable scope for each function, holding static variables, and a scope for each function call. Variables declared in a scope is generally not directly accessible by other scopes. E.g. in order to access a variable defined in the global scope, from the scope of a function body, the `global` statement can be used to create a local alias variable pointing of the global variable. Similarly

the static scope can be access with a `static` statement. All blocs share the scope of their parent.

The only variables directly accessible in both function and global scope, are the *super-globals*. They are variables, containing arrays, including `$_POST`, `$_GET`, `$_REQUEST` and `$_COOKIE` for fetching data from HTTP POST requests, HTTP GET requests, HTTP POST and GET requests, and cookies respectively, `$_SERVER` and `$_ENV` for accessing server settings and environment variables, and a few others. Program 2.1 shows an example of using *superglobals*.

►Other super-globals◄

Program 2.1 Global variables used in function scope

```
1 session_start();
2 $username = "";
3
4 function setUser($ID, $name) {
5     global $username;
6     $_SESSION["ID"] = $ID;
7     $username = $name;
8 }
9
10 setUser(1, "Admin");
11
12 echo "Hello $username $_SESSION['ID']"; // Result:
    Hello Admin 1
```

While the previous superglobals were defined by external conditions, such as requests or server settings, the variable `$GLOBALS` is an array modeling the global scope. Reading, modifying, and adding entries in this array are equivalent to reading, modifying, and initializing variables in the global scope, and can thus be used to access global variables in function scope without using the `global` keyword.

2.2.3 References

Many languages have a notion of pointers and provides the ability for variables to be a pointer to some value. In PHP the concept of references can easily be confused with pointers, however references are not pointers. Variables names and their content is treated as different things in PHP, meaning that variable names are in fact just names for a specific content. Making a reference in PHP corresponds to giving the same content another name. Assigning a variable that is already a reference as a reference of another variable will remove the binding to the original content and bind the variable to the new content. The PHP concept of references also mean that it is not possible to change a reference by a reference as shown in program 2.2.

Knowing how PHP handles names and content as two different concepts assignment can be seen as copying the content and assigning this new content a name. In practice, a copy-on-write strategy is employed which increases performance and decreases memory usage compared with a naive copy-on-assign

Program 2.2 Overwriting references

```
1 $hello = "world";
2 $hello2 = "stupid";
3
4 function change(&$input) {
5     $input = &$GLOBALS["hello2"];
6     $input = "awesome";
7 }
8
9 change($hello);
10 // Result: $hello2 = "awesome" and $hello remains
    unchanged
```

strategy. In program 2.3 after line 3 both **\$a**, **\$b**, and **\$c** point to the same array. After evaluating line 4 the array is copied, updated and **\$b** now points to the new array. Meanwhile **\$a** and **\$c** are still pointing to the same array since none of them has changed from the original array. In the example only two copies of the array are ever stored whereas a blind copy-on-assign strategy would have stored three copies of which two would never differ.

Program 2.3 Copy-on-write strategy

```
1 $a = [1,2,3];
2 $b = $a;
3 $c = $b;
4 $b[1] = 5;
5 echo $a[1] . ", " . $b[1] . ", " . $c[1];
6 // Result: 2, 5, 2
```

As an alternative PHP offers an explicit way to assign and pass function parameters by-reference. Using the ampersand operator multiple variables can reference the same value as shown in program 2.4

Program 2.4 Aliasing

```
1 function byvalFunc($input) {
2     $input["hello"] = "PHP";
3 }
4
5 function byrefFunc(&$input) {
6     $input["hello"] = "PHP";
7 }
8
9 $greet = ["hello" => "world"];
10 byvalFunc($greet);
11 echo $greet["hello"]; // Result: world
12 byrefFunc($greet);
13 echo $greet["hello"]; // Result: PHP
```

PHP arrays are treated as ordinary values and are thus copied like other values when assigning variables. The copy is deep in that the inner-arrays of

multi-dimensional arrays will be copied as well. There is however one exception to the deep-copy of arrays namely that references inside arrays are kept even after copying. This effect can be seen in program 2.5.

Program 2.5 Keeping references in arrays

```
1 $a = [1,2,3];
2 $c = &$a[0];
3 $b = $a;
4 $c = 5; // Result: $a = [5,2,3]; $b = [5,2,3]; $c = 5;
5 $b[0] = 6; // Result: $a = [6,2,3]; $b = [6,2,3]; $c =
6           6;
7 $a[0] = 7; // Result: $a = [7,2,3]; $b = [7,2,3]; $c =
8           7;
9 $b[1] = 8; // Result: $a = [7,2,3]; $b = [7,8,3];
```

►write about monotone framework◀

Chapter 3

Dynamic Analysis

The PHP array data structure allows for many different kinds of use ranging from lists over maps to trees and tables. The purpose of the dynamic analysis in this chapter is to detect patterns of array-usage in order to be able to identify some more restrictive data types contained within the way PHP arrays are used. The results of the analysis are used to choose a suitable abstraction for the static analysis in chapter 4. The first section of this chapter describes basic definitions used in the dynamic analysis to detect patterns. With the definitions in place a hypothesis for the analysis is formed in section 3.1 followed by a discussion of implementation details in section 3.2. Section 3.3 presents the results of the dynamic analysis and section 3.4 draws the conclusion of the analysis.

Definition 1. Let a be an array containing values of the same type, where all integer keys from 0 to $\text{count}(a) - 1$ exists. Then a can be considered an array of type list.

An example of an array used as a list can be seen in program 3.1, where an element is appended to the list and shifted off the beginning of the list. The values of the `$numbers` array all share the same type (integers) and the keys, though never directly manipulated, range at initialization from 0 to 2. The following operations all preserve the type consensus of the values and the type of the keys.

Program 3.1 Array used as a list

```
1 $numbers = [1,2,3];  
2 $numbers[] = 4; // $numbers = [1,2,3,4]  
3 $first = array_shift($numbers); // $numbers = [2,3,4]
```

Besides the `array_shift` function and the array append operation, `v[]`, the PHP library contains many other library functions for manipulating lists, e.g. `array_push`, `array_pop`, `sort`, etc.

Arrays can also explicitly define keys, which can be either a string or an integer.

Definition 2. Let a be an array containing values of the same type, where some integer key from 0 to $\text{count}(a) - 1$ does not exist. Then a can be considered an array of type map.

As the name suggests, maps can be used as a mapping from a string/integer to a value. In the dynamic analysis a map from pure integer keys is also denoted a sparse list. In program 3.2 the `$text_to_int` array is a mapping from strings containing some numbers to its corresponding integer representation.

Program 3.2 Array used as a map

```

1 $text_to_int =
2   [
3       'one' => 1,
4       'two' => 2,
5       'three' => 3
6   ];
7 echo $text_to_int[$input];
8 $keys = array_keys($text_to_int);
9 // $keys = ["one", "two", "three"]
10 $values = array_values($text_to_int);
11 // $values = [1,2,3]
```

If given a map-array, the values or keys can be fetched using the functions `array_values` or `array_keys` respectively. These functions return an array of the list type.

Finally arrays can be treated as objects, i.e. the entries can be viewed as properties of arbitrary type. These arrays could be replaced by the `stdClass` which mainly is used for its dynamic properties, just like arrays. Some of the build-in arrays of PHP can be considered objects, including the `$_SERVER` array¹. This array contains server and execution environment information, which are of different types, e.g. `$_SERVER['argv']` is an array of arguments passed to the interpreter, and `$_SERVER['REQUEST_TIME']` is an integer UNIX timestamp of the start of the request.

Definition 3. Let a be an array containing values of different types. Then a can be considered an array of type object.

3.1 Hypothesis

We hypothesize that any given array throughout its lifespan from initialization to last usage can be viewed as one, and only one, of the above mentioned types, i.e. either as a list, a map or an object. It is also expected that the arrays in general are acyclic and that *append*, *push*, *pop*, *shift*, and *unshift* operations are only used on arrays of the list type.

If the hypothesis holds it should be possible to statically analyse the code to identify these types and detect errors related to misuse of the arrays e.g. using maps as lists or vice versa. The hypothesis is tested against a corpus consisting

¹<http://php.net/manual/en/reserved.variables.server.php>

of ten widely used open source frameworks, by performing a dynamic analysis of the code.

The frameworks chosen all implement some kind of test suite written in PHPUnit², a unit testing framework for PHP programs similar to JUnit for Java programs. By running the test suites on a modified PHP interpreter³, we are able log and later analyze the structure and usage of the arrays. By using test suites instead of manually inspecting the frameworks through e.g. a browser, the aim is to gain a higher code coverage. This follows from the assumption that the developers are using code coverage as a metric of the quality of the test-suite. The corpus consists of the following open source frameworks:

- *WordPress*: A blogging system and a content management system [?].
- *phpMyAdmin*: An administration panel for managing MySQL databases [?].
- *MediaWiki*: A framework for creating knowledge base sites like Wikipedia [?].
- *Joomla*: A content management system [?].
- *CodeIgniter*: A lightweight framework for building web applications [?].
- *phpBB*: A forum platform [?].
- *Symfony 2*: A framework used in many major systems, such as phpBB, Magento and Drupal [?].
- *Magento 2*: An e-commerce platform [?].
- *Zend Framework*: A framework for web development focused on simplicity, reusability and performance [?].
- *Part*: A lightweight content management system developed by one of the authors of this thesis [?].

3.2 Implementation

This section contains the implementation details of the dynamic analysis. The last part of the section discusses flaws and limitations arisen from the chosen implementation of the analysis. The analysis consists of two phases: *test suite execution* with logging of feature usage and *analysis* of the logged data. The test suites are executed on a modified version of the official PHP interpreter to enable logging of feature usage.

²<https://phpunit.de/>

³<https://github.com/Silwing/php-src>

3.2.1 Logging of feature usage

All usages of array reads, array writes, assignments, array library functions (`array_push()`, `array_pop()`, `array_search()`, `count()`, etc.), and every array initialization are logged while executing the test suites. These are logged in a CSV file where each line is a log entry containing information separated by the tab character. All entries begin with *line type*, which identifies the type of the entry. The possible line types are described in the list below.

- *array function*: Every call to a library array function⁴, such as `count()` or `array_push()` are logged with the function name as line type. The array functions do not include the `array()` used to initialize an array, since it is a language construct and not an actual function. Some subroutines are also logged when dealing with multiple arrays in the same function, e.g. `array_mr_part` used by the `array_merge` function.
- *array_read*: Every read from an array is logged with the array being read from and the key used as well as the type of the value being.
- *array_write*: All array writes of the form `$x[$key] = $y` are logged with the array being read from (`$x`) the key (`$key`).
- *array_append*: When elements are added to the array using the append method, `$x[] = $y`, this is logged with the array being read from, `$x`.
- *assign_**: Every assignment is logged as either `assign_var`, `assign_tmp`, `assign_const` or `assign_ref` depending on whether the value being assigned is a constant, temporary variable, variable or reference respectively. The assignment `$x = (string) $y` is an example of an assignment from a temporary variable. Here the `$y` variable is casted and saved in a temporary variable which then is assigned to `$x`. One of these lines always follow `array_write` and `array_append` and is used to determine the type of value written in those lines.
- *array_init*: When an array is initialized without the static keyword, using either the `array('key' => 'value')` construct or the corresponding bracket notation, `['key' => 'value']`, it is logged with the array being created. If the array is initialized in any other way, e.g. as a field or by array write it is not logged with `array_init`.
- *hash_init*: Whenever a hash table, the underlying structure of arrays, is initialized, this is logged with the memory address of the table.

Arrays are logged as a tuple with four entries, (t, d, s, a) , where $t \in T \times C$ is the type of the array, d is the depth of the array, s is the size of the array, and a is the memory address. Here $T = \{\text{List, Map, Sparse List}\}$ and $C = \{\text{Cyclic, Acyclic}\}$. The array type logged indicates the type of keys present

⁴<http://php.net/manual/en/ref.array.php>

1	hash_init	0x539				
2	array_init	1	a.php	0	1	0x539
3	assign_tmp	1	a.php	NULL	array	1
	3	0x539				
4	array_append	2	a.php	1	1	0x539
	long	4				

(3.1.1) Log from running file **a.php**

```

1 $a = [1,2,3];
2 $a[] = 4;

```

(3.1.2) File: **a.php**

Figure 3.1: Example of the result from a run with the modified interpreter.

in the array, see definitions 1 and 2, as well as whether it contains any self-references. Any array with a self-reference is **Cyclic** and all other arrays are **Acyclic**.

Objects are logged as their instance name, integers and floats are logged with their value, string are logged only as **string**, booleans as 0 if **false** otherwise 1, and the null value logged as **NULL**. Strings are is generally not logged with a value, because doing so increases the file-size drastically, tends to corrupt the file when containing binary data, and has not proven necessary for the analysis.

All entries besides the **hash_init** entries contain a line number and a file path to where the action occurred.

3.2.2 Identification of arrays

In order to analyse how arrays are used throughout a log file, some method for identifying which arrays are mentioned on a given line is needed. E.g. in the output depicted in figure 3.1.1 all lines concerns the same array with memory address **0x539**. Here, identifying these arrays as the same is a matter of checking the address. Due to the size of the test suites, relying only on the address is not a stable approach. Over time addresses will be reused and false identifications will happen. This issue can be solved by depending on the **hash_init** line, which indicates that a new array is initialized at some address. If such a line occurs between two usages of an address they can not automatically be considered representing the same array.

Since the log files are generated from running a test-suite, relying on addresses for identification alone might result in a skewed analysis with an over-representation of arrays occurring in *critical* code. Determining array equality based on initialization location in the file should provide a more equal representation of arrays, not letting some arrays dominate the statistics. Locational identification would however identify four different arrays in example 3.1.1, which does not reflect the program 3.1.2 and thus the address is still needed to identify the same array across multiple code locations.

Definition 4. Given two lines, l_1 and l_2 , from a log file, R , as described above, each containing an array $x_1 \in l_1$ and $x_2 \in l_2$, where $x_1 = (t_1, d_1, s_1, a_1)$ and $x_2 = (t_2, d_2, s_2, a_2)$, the arrays are said to be positional equal, $x_1 \stackrel{pos}{=} x_2$, if and

only if the two lines share the same line number, line type, and file or $a_1 = a_2$ and there is no

hash_init a_1

line between l_1 and l_2 .

This definition utilizes file position and addresses in order to identify arrays. The line type has been added in order to heighten precision, since multiple different operations, on different arrays, may occur on the same line.

Definition 5. Given two lines $l_1, l_2 \in R$ and two arrays, $x_1 \in l_1$ and $x_2 \in l_2$, then id is a ID-function if and only if

$$id(x_1) = id(x_2) \Leftrightarrow x_1 \stackrel{pos}{=} x_2$$

When iterating through a log file from top to bottom IDs can be generated by keeping a mapping from locations to IDs and from addresses to IDs, and by *forgetting* addresses when a **hash_init** line is observed.

3.2.3 Determining type

Determining the type of every positional distinct array is done by first determining the key type, $t \in T$, then determining the type of the values, and from this deduce the type as either List, Map or Object as defined in the beginning of this chapter. Since an array can change type during a program, the type of the keys is more accurately a set of types. If an array is observed with different key types throughout the analysis, then the key type of the array is the set of these types. E.g, let an array be observed at one point with type List and later with type Sparse List, then the type of the array is {List, Sparse List}.

Detecting the key type for each array is done by traversing the file from top to bottom inspecting each line. If a line contains an array a the type of the line is associated with the corresponding ID of the array.

Detecting the type of values is also done by traversing through the log file from top to bottom. Here the reads and writes from and to the arrays are used to determine the types of the values. This is done by associating all the types of the values read/written with the respective array.

For every array with key and value type information, it is now possible to determine whether it is a list, map, object or uncategorizable. Given an array, a , with type information, if a has multiple key types, it is considered uncategorizable. Else if a contains values of a single type, it is either a map or a list, depending on the key type. If the values have multiple types, then a is an object.

When the types of the arrays are determined, we can analyse the operations used with each type of array. This is done by once again iterating through the log file and associating line types with the type of the arrays.

3.2.4 Compiling and running PHP with logging

For the purpose of the dynamic analysis, Vagrant⁵ is used to create a clean and reproducible environment. A Vagrant initialization file⁶ is used to setup a virtual machine running a 64-bit Ubuntu 14.04, install the necessary dependencies and compile the modified PHP Interpreter. The environment for running the corpus test suites is then ready and can be accessed via SSH on the virtual machine. The folder `/vagrant/corpus` contains a Makefile which can be used to fetch dependencies and corpus frameworks as well as running all the test suites.

All modifications to the interpreter are guarded by an ini-directive[?, Chapter 14.12] that is disabled by default when compiling and running the modified interpreter source. The logging can be enabled via `php.ini` or for single runs as seen in figure 3.2

```
1 $ php -d rb.enable_debug=1 -d rb.enable_debug_file=<
    path-to-log-file > <path-to-php-file >
```

(3.2.1) Enable logging for running a single PHP file.

```
1 rb.enable_debug=1
2 rb.enable_debug_file="/path/to/output/csv/file "
```

(3.2.2) Enable logging in php.ini.

Figure 3.2: How-to enable logging

3.2.5 Limitations

Since the analysis is performed by a modified interpreter, there are some imposed limitations on the achievable precision.

- `array_init` does not capture the type of the array at initialization. This implies that the type of arrays initialized without being assigned or manipulated afterwards are not captured by the analysis. In figure 3.3.2 line 3, the call to `print_r` does yield an `array_init` line in the log 3.3.1 but with no type, depth 1 and size 0. The size should be 3 and the type List.
- Detecting the type relies on the type of the values read or written to the arrays. This implies that there is no reasoning about entries or arrays never read or written. This might produce some false positives.
- Callables can be written as anonymous functions, strings, or arrays, containing an instance and a string. This analysis will fail to classify arrays containing callables, initialized differently, correctly, thus introducing false negatives. E.g in example 3.3; `$callable1`, `$callable3` and `$callable4` are all valid callables, however `$callable2` is not, since `$a->f2()` is a private function.

⁵<http://vagrantup.com/>

⁶<http://github.com/Silwing/tapas-survey>

Program 3.3 Callables in PHP

```
1 class A{
2
3     public function f1(){
4         ...
5     }
6
7     private function f2(){
8         ...
9     }
10
11 }
12
13 function f(){
14     ...
15 }
16
17 $a = new A();
18
19 $callable1 = [$a, "f1"];
20 $callable2 = [$a, "f2"];
21 $callable3 = "f";
22 $callable4 = function() use ($a){
23     ...
24 };
```

- Multiple operations of the same type on different arrays leads to sharing of IDs. Following the limited information available to distinguish operations, operations such as assign to an multidimensional array leads to sharing it between the array and its sub-arrays, (see figure 3.3.2 line 5). This follows from the value first being assigned to the sub-array which then is assigned to the super-array. This leads to false negatives and could be solved if the interpreter kept character location information in addition to line number and file name.

3.3 Results

Table 3.1 shows that across all frameworks in the corpus less than 1% (column 5) of the arrays are detected as being cyclic. Cyclic arrays can only be created by using the the PHP reference operator, `&`, which must be used explicitly. Since PHP by default copies values on assignment cyclic arrays can not be created by simple assignments. The largest amount of cyclic arrays detected in the corpus is in PhpMyAdmin with a total of 10 cyclic arrays out of 3,373 identified arrays. By assuming that arrays are acyclic, the static analysis would not have to take recursive types into consideration. Since so few cyclic arrays were found a manual inspection of the code locations of the findings was possible. The manual inspections showed that almost all of the reported cyclic arrays are uses of the superglobal `$GLOBALS` which is an array consisting of all variables

```

1 hash_init      0x2A
2 array_init     3      b.php      0      1      0      0x2A
3 hash_init     0x2B
4 array_write    5      b.php      0      1      0      0x2B  long
   1 NULL
5 hash_init     0x2C
6 array_write    5      b.php      0      1      0      0x2C  long
   2 NULL
7 assign_const   5      b.php      NULL   long   3

```

(3.3.1) Log from running file **b.php**

```

1 <?php
2
3 print_r([1,2,3]);
4
5 $a[1][2] = 3;

```

(3.3.2) File: **b.php**

Figure 3.3: A problematic program

defined in the global scope. This array also has a reference to itself which makes it a cyclic array. Only two of the frameworks in the corpus had cases where it was not immediately clear whether it was a use of the **\$GLOBALS** array: Zend Framework 2 and Symfony, these are noted in column 4 as “NG Cyclic” arrays. These few cases are function input parameters reported as cyclic which might in fact be the **\$GLOBALS** array passed to the function however it is not confirmed to always be the case. The **\$GLOBALS** array is a special case to handle even without taking its cyclic structure into account.

Based on the results of the manual inspection the hypothesis of arrays being acyclic is true for almost all frameworks in the corpus.

Framework	# Arrays	# Cyclic	# NG Cyclic	%
Code igniter	331	0	0	0.0%
Joomla	1969	2	0	0.1%
Magento2	6942	0	0	0.0%
Mediawiki	27368	1	0	<0.1%
Part	378	0	0	0.0%
phpBB	2529	1	0	<0.1%
PhpMyAdmin	3373	10	0	0.3%
Symfony	3707	6	6	0.2%
Wordpress	3054	1	0	<0.1%
Zend Framework 2	4381	3	2	0.1%

Table 3.1: Amount of cyclic arrays detected in the corpus

Figure 3.4 shows the distribution of array types for the frameworks. Between 4% and 12% of the arrays are uncategorizable. These include false uncategorizables originating from flaws in the array identification. If multiple categorizable arrays from different categories are identified as a single array the combined array possibly ends up in the uncategorizable part of the distribution.

The Object group marked with List is by definition categorized as objects, but they might fit better into the List category, as lists of a top level type. Whereas string writes are good predictors for map type arrays and appending and list array functions are good predictors for list type arrays, no predictors have been found for the object type arrays. Without any predictors for an object-like array type it is not possible to define uses and misuses of such a structure. Instead the object-like array type can be absorbed partly by the list type and partly by the map type.

	List	Map	Sparse List	Object	Object (L)	Object (SL)	Uncategorizable
Code Igniter	39.66%	36.21%	2.59%	12.93%	2.59%	0.00%	6.03%
Joomla	30.78%	39.13%	2.17%	20.02%	3.66%	0.11%	4.12%
Magento 2	23.54%	46.51%	3.38%	17.93%	2.63%	0.70%	5.30%
MediaWiki	32.48%	32.23%	2.69%	15.60%	8.20%	0.49%	8.32%
Part	33.33%	39.10%	0.00%	12.82%	5.77%	0.00%	8.97%
phpBB	27.13%	33.33%	3.17%	25.11%	4.33%	0.14%	6.78%
PhpMyAdmin	33.24%	33.43%	2.09%	14.06%	5.89%	0.38%	10.92%
Symfony	34.32%	28.01%	1.99%	14.86%	8.63%	0.21%	11.99%
WordPress	35.50%	33.03%	2.02%	14.11%	6.61%	0.45%	8.29%
Zend Framework 2	30.99%	35.07%	1.08%	19.78%	6.25%	0.00%	6.82%

Table 3.2: Distribution of different array types

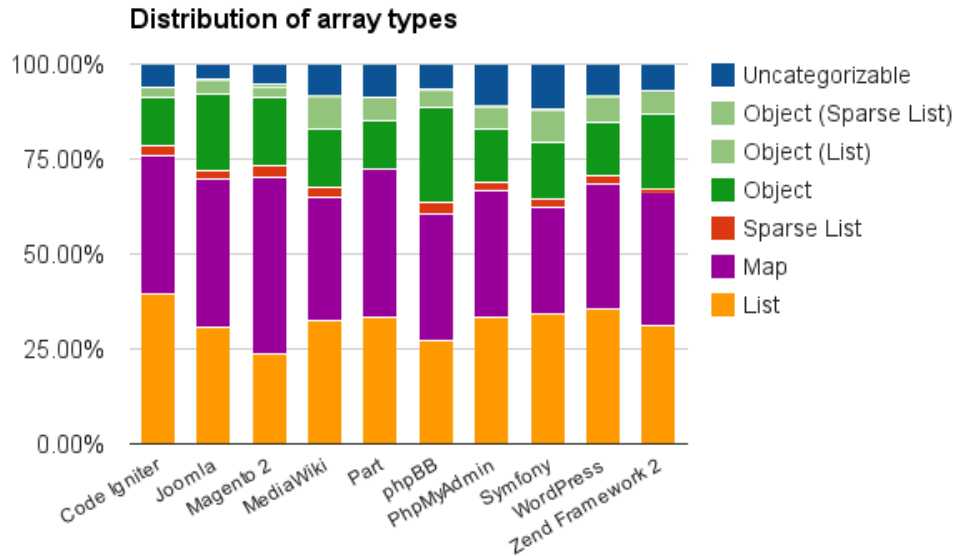


Figure 3.4: Distribution of different types of arrays

Figure 3.5 shows the distribution of operations on the arrays over the different array types from figure 3.4. Write and append correspond to the language features for writing to arrays:

```
1 $a = [];
```

```

2 $a[0] = 42; // array write
3 $a[] = 1337; // array append

```

These operations are by far the most used. The built-in push function is equivalent to the append operation if given only a single argument. The documentation recommends the append operation in such situations for performance reasons, which aligns with the use of append over push in the figure. The operation on arrays of map and object type consists almost entirely of write operations, whereas arrays of type list have some write operation but mostly append operations. This indicates that append is a good predictor for arrays in the list category.

The distribution of operations support the claim that the List-marked objects fit better into the list category than the object category, since multiple list operations are frequently used with these arrays.

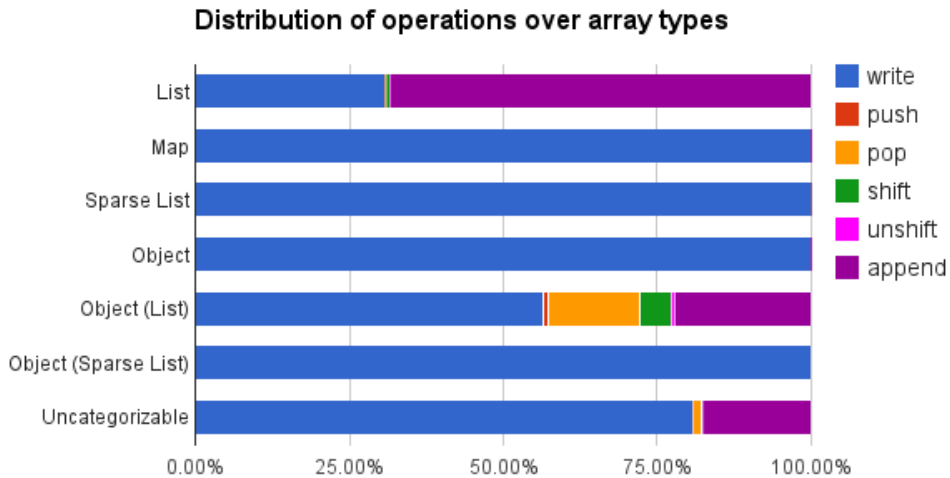


Figure 3.5: Distribution of array-changing operations over array types

3.4 Conclusion

The purpose of the dynamic analysis was to determine whether PHP array usage can be split into semantic categories and if arrays stay in one category during their lifetime.

The results show that generally arrays keep the same type during their lifetime. Furthermore, the usage of specific array operations proved almost exclusive for lists. This information can be used to define unexpected behavior reported by the static analysis.

A significant amount of arrays turned out to be objects with list-keys by the initial definition. These objects indicate that the object type is not providing significant information in itself, and show the possibility that the definitions of

maps and lists consume the object type, i.e. letting maps and lists allow values of different type.

Almost every framework in the corpus contains cyclic arrays, hence the static analysis has to take the possibility of recursive types into consideration. However, the small amount of cyclic arrays indicate that an imprecise approach will have a minimal impact on the overall precision.

Chapter 4

Data Flow Analysis

Based on the findings in chapter 3 a dataflow analysis is designed and described in this chapter. The aim of the analysis is to introduce two new array types, array-lists and array-maps respectively, and subsequently report any suspicious behaviour on these arrays. Examples of suspicious behaviour could be appending on maps, using keys of type string on lists, adding elements of a different type to an array, etc.

While running the analysis on the corpus of the dynamic analysis would be ideal, limited time forces the analysis to be defined on a subset of PHP, introduced in section 4.1. The analysis is preformed using the monotone framework. An instance of this framework is derived, in section 4.7, from a control-flow-graph, introduced in section 4.2, a model of program states, introduced in section 4.3, and finally transfer functions for each CFG-node, introduced in section 4.4. Since PHP rely heavily on coercion, a seperate section (sec. 4.5) covers coercion of abstract values. Section 4.6 covers abstract evaluation and the implementation of the analysis is briefly introduced in section 4.8.

4.1 PHP Language subset

To simplify the static analysis a subset of the PHP language, P0, is used. The PHP language has no formal definition and thus the language used here can not be formally shown to be a subset of the complete PHP language. The full PHP language is defined by the reference Zend Interpreter.

To simplify the analysis and keep the focus on arrays; resource handles and objects have been completely removed from the language. Dynamic dispatch (variable function names), variable variables and dynamic loading of code (**require** and **include**) have also been omitted. It is assumed that any possible path in a function body results in a return statement and return statements are only allowed in a function body. There are no anonymous functions, a limited number of statements but the language does support all reference features, i.e. reference assigning entries in an array or variables. Reading from and writing to the **\$GLOBALS** array should get or change the value of the variables, respectively. Creating a new variable by adding an new entry to the array does however not initialize a new variable in the global scope.

The syntax can be expressed with grammar 4.1. Here $e : \langle expr \rangle$ denotes an expression, $e : \langle rexpr \rangle$ a reference expression, and $e : \langle vexpr \rangle$ a variable expression. Furthermore it being a subset of PHP, a program is only valid in P0 if it is also a valid PHP program. E.g. while the syntax allows negation of arrays, this action yields a fatal error in PHP and may be considered as an invalid program, hence also an invalid P0 program.

Notice that returning an $\langle rexpr \rangle$ with a non-reference function might result in a fatal error, e.g. when returning the result of an array-append operation, but the same operation is valid in a reference-function.

►elaborate◄

►We are not supporting initializing an array with references◄

►Write about support of alias◄

►No string array access◄

►Only supporting operation signatures mentioned in abstract evaluation◄

4.2 Control flow graph

Given a P0 program, $p : \langle program \rangle$, parsed to a abstract syntax tree, the control flow graph can be constructed recursively. Most of the nodes takes arguments. Here h denotes variables of type HeapTemps, t variables of type Temps, and c variables of either type. Subgraphs are denoted by a grey rectangles with letters $\llbracket S \rrbracket$, $\llbracket E \rrbracket(t)$, $\llbracket R \rrbracket(h)$, $\llbracket V \rrbracket(h)$, and $\llbracket T \rrbracket(c)$ which denotes subgraph of statement S , expression E with the resulting value stored in variable t , reference expression R with resulting heap-location-set stored in h , variable expression V with resulting heap-location-set stored in h , or either expression with the result stored in c respectively. For a given argument, c , to the subgraph, the variable corresponds to the target variable, c_{tar} , in the subgraph. E.g. let $E = 1+(2+3)$ then the graph of $\llbracket E \rrbracket(t)$ is recursively constructed as in graph 4.1. Notice how the argument, t , passed to the graph is the third argument of the last $bop_+(_)$ node.

Definition 6. A control-flow-graph $G = (V, E, s, t)$ where V is a set of nodes, E is a set of node pairs representing an edge between two nodes, $s \in V$ is a start node and $t \in V$ is an exit node. When illustrated as a graph, the entry node is marked with an ingoing edge with no origin and the exit node is marked with an outgoing edge with no target. E.g. in example graph 4.1, $s = constRead(1, t_1)$ and $t = bop_+(t_1, t_2, t)$.

There are seventeen different nodes, all introduced below

start: This node indicates the start of a program and is the first node of any program or function body.

$bop_{\oplus}(t_l, t_r, t_{tar})$, $sop_{\oplus}(t_l, t_r, t_{tar})$, $uop_{\circ}(t, t_{tar})$, $inc_{\circ}(h, t_{tar})$: These nodes indicates binary, short-circuit-binary, uanry, and increment/decrement operations, respectively. The operations of the first three are all performed on temporary storage, while the forth is performed on the heap. The last argument, t_{tar} , indicates where the result of performing the operation

$$\begin{aligned}
\langle \text{program} \rangle &::= (\langle \text{function-definition} \rangle \mid \langle \text{statement} \rangle)^* \\
\langle \text{function-definition} \rangle &::= \text{'function' } \langle \&? \langle \text{function-name} \rangle \text{' (} \langle \&? \langle \text{var} \rangle \text{ (' , ' } \langle \&? \langle \text{var} \rangle \rangle^* \text{) } \mid \epsilon \rangle \text{' } \langle \text{block} \rangle \\
\langle \text{statement} \rangle &::= \text{'while' } \langle \text{C' } \langle \text{expr} \rangle \text{' } \rangle \langle \text{statement} \rangle \\
&\mid \text{'for' } \langle \text{C' } \langle \text{expr} \rangle^? \text{' ; ' } \langle \text{expr} \rangle^? \text{' } \rangle \langle \text{statement} \rangle \\
&\mid \text{'if' } \langle \text{C' } \langle \text{expr} \rangle \text{' } \rangle \langle \text{statement} \rangle \\
&\mid \text{'if' } \langle \text{C' } \langle \text{expr} \rangle \text{' } \rangle \langle \text{statement} \rangle \text{'else' } \langle \text{statement} \rangle \\
&\mid \text{' ; ' } \\
&\mid \langle \text{expr} \rangle \text{' ; ' } \\
&\mid \text{'global' } \langle \text{var} \rangle \text{' (' , ' } \langle \text{var} \rangle \rangle^* \text{' ; ' } \\
&\mid \text{'return' } (\langle \text{expr} \rangle \mid \langle \text{rexpr} \rangle)^? \text{' ; ' } \\
&\mid \langle \text{block} \rangle \\
\langle \text{expr} \rangle &::= \langle \text{expr} \rangle \oplus \langle \text{expr} \rangle \\
&\mid \circ \langle \text{expr} \rangle \\
&\mid \langle \text{C' } \langle \text{expr} \rangle \text{' } \rangle \\
&\mid \langle \text{vexpr} \rangle \text{' + ' } \\
&\mid \langle \text{vexpr} \rangle \text{' - ' } \\
&\mid \text{' + ' } \langle \text{vexpr} \rangle \\
&\mid \text{' - ' } \langle \text{vexpr} \rangle \\
&\mid \langle \text{var} \rangle \\
&\mid \langle \text{expr} \rangle \text{' [' } \langle \text{expr} \rangle \text{'] ' } \\
&\mid \langle \text{function-reference} \rangle \\
&\mid \langle \text{const} \rangle \\
&\mid \langle \text{assignment} \rangle \\
&\mid \text{' [' } (\epsilon \mid \langle \text{array-init-entry} \rangle \text{' (' , ' } \langle \text{array-init-entry} \rangle \rangle^* \text{'] ' } \\
&\mid \text{'array' } \langle \text{C' } (\epsilon \mid \langle \text{array-init-entry} \rangle \text{' (' , ' } \langle \text{array-init-entry} \rangle \rangle^* \text{') ' } \\
\langle \text{function-reference} \rangle &::= \langle \text{function-name} \rangle \langle \text{C' } (\langle \text{function-arg} \rangle \text{' (' , ' } \langle \text{function-arg} \rangle \rangle^* \mid \epsilon) \text{' } \rangle \\
\langle \text{function-arg} \rangle &::= \langle \text{expr} \rangle \\
&\mid \langle \text{rexpr} \rangle \\
\langle \text{rexpr} \rangle &::= \langle \text{var} \rangle \\
&\mid \langle \text{function-reference} \rangle \\
&\mid \langle \text{rexpr} \rangle \text{' [' } \\
&\mid \langle \text{rexpr} \rangle \text{' [' } \langle \text{expr} \rangle \text{'] ' } \\
\langle \text{vexpr} \rangle &::= \langle \text{var} \rangle \\
&\mid \langle \text{rexpr} \rangle \text{' [' } \\
&\mid \langle \text{rexpr} \rangle \text{' [' } \langle \text{expr} \rangle \text{'] ' } \\
\langle \text{array-init-entry} \rangle &::= \langle \text{expr} \rangle \text{' => ' } \langle \text{expr} \rangle \\
&\mid \langle \text{expr} \rangle \\
\langle \text{assignment} \rangle &::= \langle \text{rexpr} \rangle \text{' [' } \text{' = ' } \langle \text{expr} \rangle \text{' ; ' } \\
&\mid \langle \text{rexpr} \rangle \text{' [' } \langle \text{expr} \rangle \text{'] ' } \text{' = ' } \langle \text{expr} \rangle \text{' ; ' } \\
&\mid \langle \text{var} \rangle \text{' = ' } \langle \text{expr} \rangle \\
&\mid \langle \text{rexpr} \rangle \text{' [' } \text{' = ' } \langle \& \rangle \langle \text{rexpr} \rangle \text{' ; ' } \\
&\mid \langle \text{rexpr} \rangle \text{' [' } \langle \text{expr} \rangle \text{'] ' } \text{' = ' } \langle \& \rangle \langle \text{rexpr} \rangle \text{' ; ' } \\
&\mid \langle \text{var} \rangle \text{' = ' } \langle \& \rangle \langle \text{rexpr} \rangle \\
\langle \text{block} \rangle &::= \text{' { ' } \langle \text{statement} \rangle^* \text{' } \text{' } \text{' }
\end{aligned}$$

should be stored. E.g. $1+2$ is a binary operation (*bop*) returning 3, this value should be stored in temporary storage at t_{tar} .

if(t): This node has one incoming and two outgoing edges, representing the choice of one branch or the other, when evaluating t .

constRead(c, t_{tar}): This node representing reading a constant into the temporary storage, at t_{tar} . The constant can be strings, booleans, null, ints etc.

varRead($\$v, c_{tar}$), *varWrite*($\v, c_{val}, t_{tar}): These nodes indicates reading from and writing to a variable, $\$v$, respectively. Depending on the context the target of the read and value of the write can either be a temporary or temporary heap storage.

arrayInit(t_{tar}): This node represents initializing an empty array in the temporary storage, at t_{tar} .

arrayAppend(t_{val}, t_{ar}), *arrayAppend*($h_{var}, c_{val}, t_{tar}$), *arrayAppend*(h_{var}, h_{tar}): With three different signatures, this node represents appending to an array, in temporary storage, in the heap, and in a read-context. Normally PHP will not allow appending without a value, but in this case, the reference of the appended entry is placed in temporary heap storage, and thus accessible for later modification. The c_{val} variable in the second node indicates that both values and references can be appended to an array.

arrayRead(h_{ar}, t_{key}, c_{tar}): Reading from an array on the heap, with value key at t_{key} . The result can either be references to the entry or values at the given key, hence the c_{tar} variable.

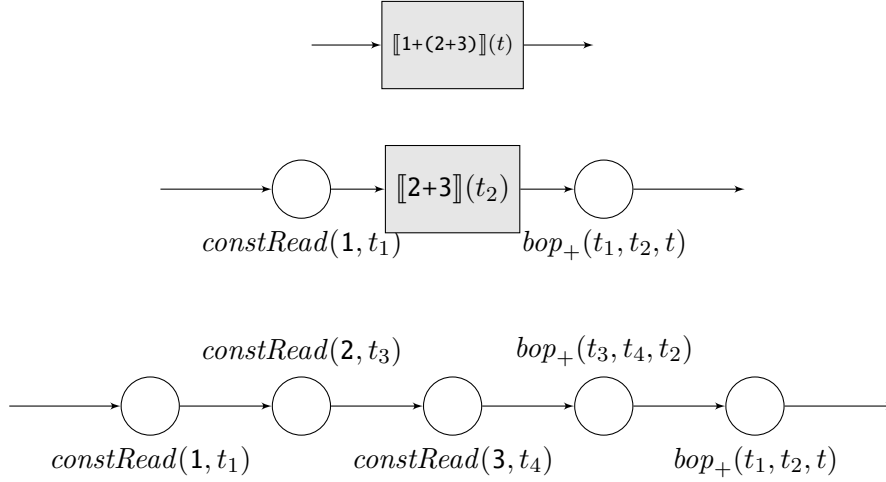
arrayWrite(t_{val}, t_{ar}), *arrayWrite*(h_{ar}, c_{val}, t_{tar}): Writing to an array in temporary storage or in the heap, respectively. Just as append, the value written can be either references or a value, hence the c_{val} variable.

call_{fn}(c_1, \dots, c_n), *exit*(c_1, \dots, c_n), *result_c*(c_{tar}): Calling a function is performed with the *call* node. This holds the name, fn , of the function being called (which can always be resolved). For every call node, c , there is a single result node, *result_c*. This restores the calling context and handles passing of the result. The node immediately before the result node is an *exit* node, which is unique to and last node of the function being called. This node holds variable names pointing to the possible values returned by the function.

nop: This node does nothing and is only there for structural purposes. It is in the control-flow-graphs denoted as a small node with no label.

The control-flow graph created from p starts with a *start* node followed by the control-flow graphs for each statement in p , connected edges from exit to entry nodes, in the order of appearance.

For each function definition encountered a separate graph is created, which may later be referenced. This graph starts with a *start* node and ends with an

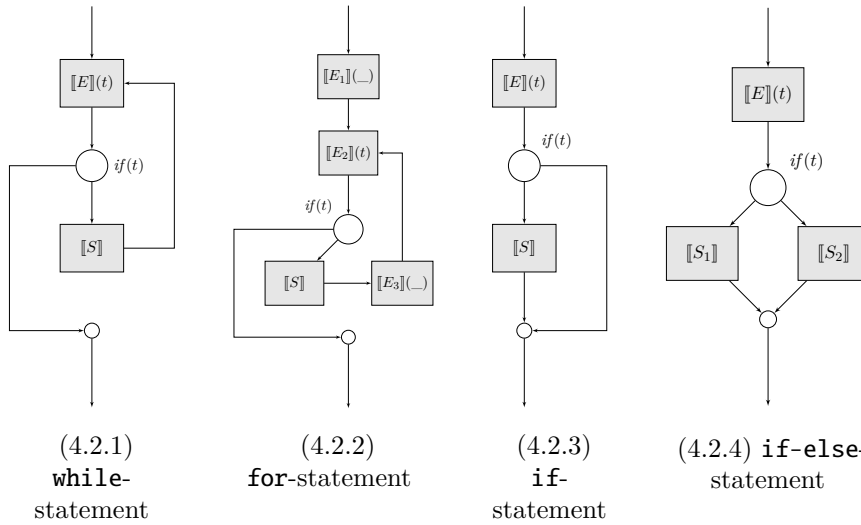


Graph 4.1: Recursively constructing graph $\llbracket 1+(2+3) \rrbracket(t)$

exit node, between these are the graph corresponding to the function body. The arguments of the exit node are the temporary variables created by the return statements.

4.2.1 Statements

Let $s : \langle \text{statement} \rangle$ be a statement, then there are nine different graphs, one for each case in the grammar 4.1. The first four, for $s = \mathbf{while}(E) S$, $s = \mathbf{for}(E_1; E_2; E_3) S$, $s = \mathbf{if}(E) S$, and $s = \mathbf{if}(E) S_1 \mathbf{else} S_2$ statements, are depicted as graph 4.2.1, 4.2.2, 4.2.3, and 4.2.4 respectively.



Return-statements, $s = \mathbf{return} T$ have three different graphs. If the return statement is empty, i.e. does not contain an expression, then this is equivalent

of returning **null**, this case yields the graph in 4.3.1. If $T : \langle reexpr \rangle$ and the function is a reference function, i.e. the function signature has an ampersand before the function name (see program 4.1.2), then the references is returned (graph 4.3.2 where $c : \text{HeapTemps}$). If this is not the case, (see program 4.1.1), assuming the program is a valid P0 program, then it is fair to assume that T can be parsed as an expression, since it cannot be an array-append operation. If it was an array append operation, then the program wouldn't be a valid PHP program and thus not a valid P0 program (see section 4.1). Assuming that the $T : \langle expr \rangle$ the graphs for a return statement should evaluate T and return the result, this is the case in graph 4.3.2 with $c : \text{Temps}$. For all graphs, the exit-node is the unique exit-node of the function.

Program 4.1 Return-statement examples

```

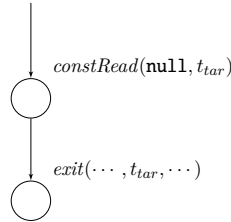
1 function normRet() {
2     $a++;
3     return $a;
4 }
5
```

(4.1.1) Normal return-statement

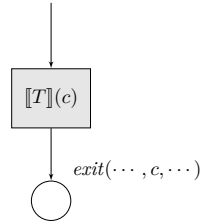
```

1 function &refRet(&$a) {
2     return $a [];
3 }
4
5
```

(4.1.2) Reference return-statement



(4.3.1)
return-statement



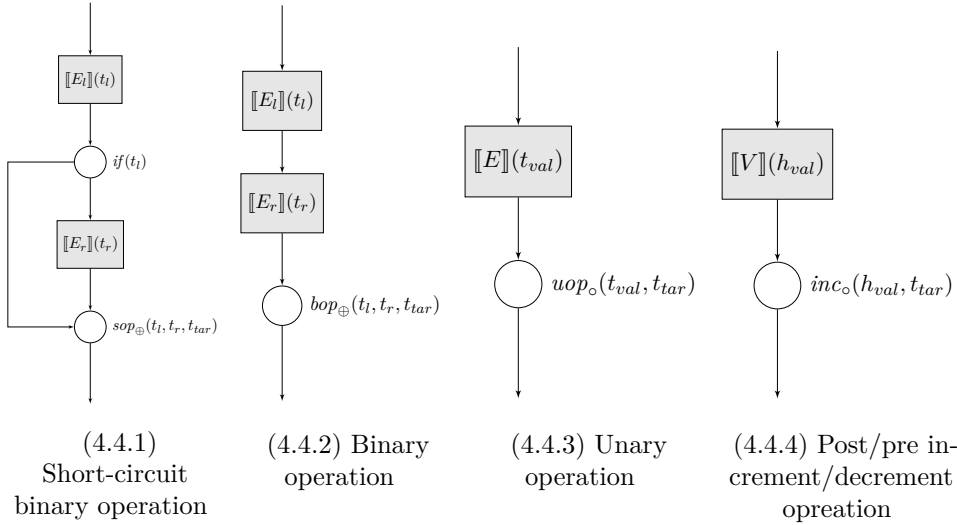
(4.3.2) return-
expression-
statement

The remaining four graphs are the empty graph, the graph of the expression statement, the graph of **global** statement, and the graph of the block statement which are all straight-foward.

4.2.2 Expressions

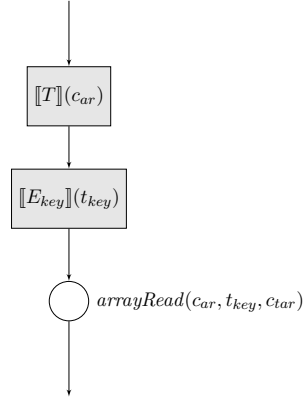
Let $e : \langle expr \rangle$ then, by ignoring the trivial parenthesized expression case, viewing the four increment/decrement operations as one, and the two array-initialization operations as one, the expression can be on nine different forms.

For $e = E_l \oplus E_r$ the graph depends on the operation. If the operation is a short-circuit operation, logical **&&** or **||**, then the graph is as 4.4.1 because only one branch may be required to be evaluated. If the operation is not a short-circuit operation then the graph becomes as 4.4.2, because both expressions must be evaluated. In both cases the operations are performed on, and saved in, the temporary storage. The unary operation, $e = \circ E$ is similar to the previous case with a graph as 4.4.3. A separate graph for unary post/pre increment/decrement operations are necessary because of the performed update on the heap location. This is the reason for the operations not being performed on the temporary storage, but instead on heap-locations directly. The result of the operation is stored in the temporary storage. Figure 4.4.4 illustrates the corresponding flow graph.

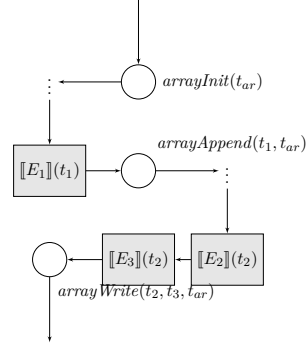


For a variable read, $e = \$v$ for some variable $\$v$, or a constant read, e.g. $e = \text{"foo"}$, the graph is a single $varRead(\$v, t_{tar})$ or $constRead(\text{"foo"}, t_{tar})$ node respectively. For an array read expression, $e = E_{ar}[E_{key}]$, the sub array expression, E_{ar} , should be evaluated before the key expression, E_{key} , and the graph then becomes like graph 4.5.1, where T is the graph corresponding to E_{ar} and $c_{ar}, c_{tar} : \text{Temps}$. If the expression is an array initialization, $e = [\dots, E_1, \dots, E_2 \Rightarrow E_3]$, then an array is first initialized in temporary storage after the entries are either appended or written to the array. Hence graph 4.5.2.

For function calls, $e = fn(T_1, \dots, T_n)$, the result variable is a temporary variable, $c_{tar} : \text{Temps}$, the arguments are either an expression, $T_i : \langle expr \rangle$ and $c_i : \text{Temps}$, or an reference expression, $T_i : \langle rexpr \rangle$ and $c_i : \text{HeapTemps}$, depending on the signature of fn . If the argument is pass-by-reference i.e. the variable, v_i is denoted with a **&** in the signature, then the expression must be a reference expression, if not, then the argument is an expression. This follows from the program being a valid P0 program. The function graph will be as graph 4.6.1, where the start node, exit node, and function body are the unique nodes of the fn function i.e. they are not copied. This ensures that the graph

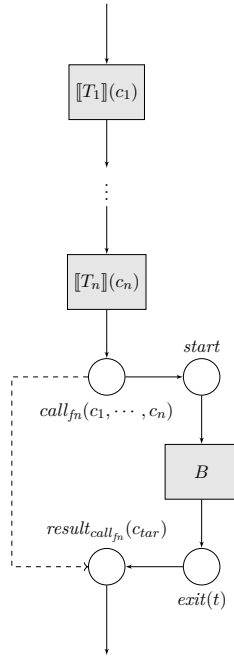


(4.5.1) Array read



(4.5.2) Array initialize

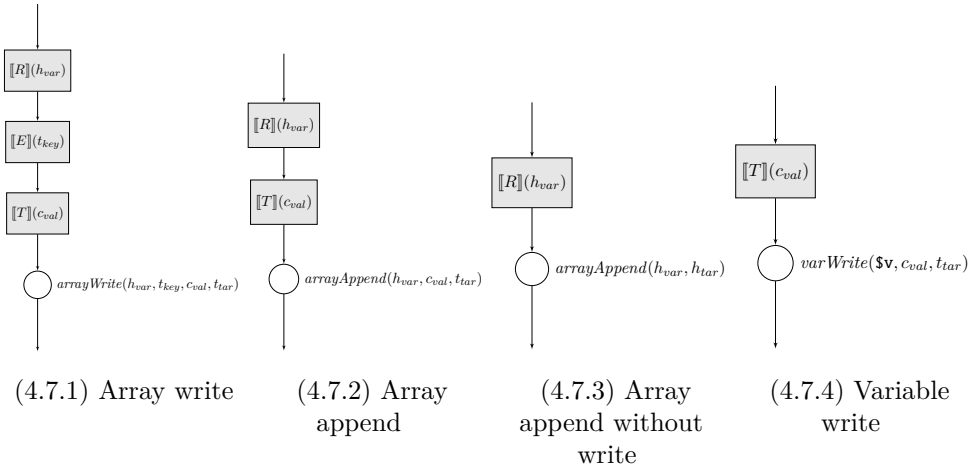
is finite, but multiple, say n , calls to the same function will yield a graph with n edges to the start node and from the exit node. Notice the dashed line going directly from the call node to the return node. This indicates that the call node may pass information directly to the return node, typically information regarding the local context before calling fn .



(4.6.1) Function call

Finally the expression can be an assignment. There are two types of assignments, a regular value-assignment and a reference-assignment, using the $\&$ operator. Each assignment can be split up in three categories, variable-, array-write-, and array-append-assignments, depending on the target (left side of the operation). Examples of these six operations can be seen in program

4.2. Due to the distinction between temporary storage of values and heap-location-sets, these six cases must be handled individually. For value array write ($e = R[E_{key}] = E_{val}$), array append ($e = R[] = E_{val}$), and variable write ($e = \$v = E_{val}$), the graphs are as 4.7.1, 4.7.2, and 4.7.4 respectively, where T is the subgraph of the value expression $E_{val} : \langle expr \rangle$ and $c_{val} : \text{Temps}$ is the temporary variable holding the result. The reference assignments, $e = R[e_{key}] = \&R_{val}$, $e = r[] = \&R_{val}$, and $e = \$v = \&R_{val}$, are similar, but with T being the subgraph of the reference expression, $R_{val} : \langle rexr \rangle$ and $c_{val} : \text{HeapTemp}$ the heap temporary variable holding the heap locations of the value.



Program 4.2 Assignments

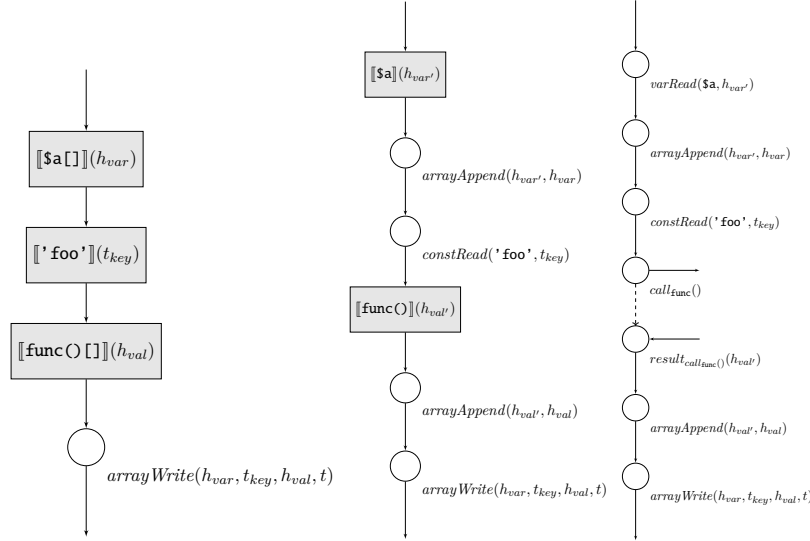
```

1 <?php
2
3 $a = []; //Value assignment
4 $a[] = 1; //Array append assignment
5 $a[1] = 2; //Array write assignment
6
7 $b = &$a; //Value reference assignment
8 $c[] = &$a; //Array append reference assignment
9 $d[1] = &$a; //Array write reference assignment

```

4.2.3 Reference and variable expressions

Since PHP supports nested assignments e.g. $\$a[]['foo'] = \&func()[]$, deciding which location to update is performed recursively. The graph of this assignment is illustrated as graph 4.8. Here the locations of variable $\$a$ is stored in variable $h_{var'}$. Then a new location, l , is appended to the array values of the locations in $h_{var'}$, and $h_{var} = \{l\}$. Now the expression of the index is resolved and stored in t_{key} , and the locations of the right side is resolved in h_{val} . Finally the locations in h_{val} are written to the array at h_{var} , i.e. l , with the key of t_{key} .



Graph 4.8: Creating graph of expression $[[\$a[]['foo']] = \&func()[]](t)$

The subtrees which takes a $h : \text{HeapTemp}$ variable are called reference- and variable-expressions. Where the only difference between the two is that reference expressions may contain function references. As seen in the previous example they resolve heap-locations rather than values. For $r = fn(c_1, \dots, c_n)$ the graphs corresponds to graph 4.6.1, with heap-locations as result, i.e. $c_{tar} : \text{HeapTemps}$. For the variable read, $r = \$v$, the graphs is a single $varRead(\$v, h_{tar})$ node. When the expression is an array-read, $r = R_{ar}[E_{key}]$, the graph corresponds to that of 4.5.1 with $c_{ar}, c_{tar} : \text{HeapTemps}$ and T being the graph of R_{ar} . Finally for the append operation, $r = R_{ar}[]$, the graphs corresponds that of graph 4.7.3.

4.3 Lattice

In order to create a inter-procedural analysis, the analysis lattice is defined as

$$\text{AnalysisLattice} = \Delta \rightarrow \text{State} \quad (4.1)$$

This is a map from a context to an abstract state, where the context, Δ consists of a list of call-sites, represented as the call nodes in the control flow graph, with length bounded by $k > 0$. The bound k must be greater than zero, since the context is chosen to decide between local and global scope.

If the context was bounded by $k = 0$ this would entail that all variables would be updated in the global scope, which with the current analysis isn't sound. In the analysis the only strong update of the heap is preformed when writing to a variable pointing to no locations, which is sound because the variable has not been initialized, hence can not have been referenced. By only using the global scope, performing an strong update with a not-null value will indicate that the variable can't be **null**. This is true for the function body, but

in the global scope the variable is uninitialized, i.e. **null**, hence the analysis would be unsound.

$$\Delta = \text{CallNode}_*^{\leq k} \quad (4.2)$$

The abstract state is a product lattice with five factors. The first two models the scope, the third the heap, and the last two models storage for intermediate results.

$$\text{State} = \text{Locals} \times \text{Globals} \times \text{Heap} \times \text{Temps} \times \text{HeapTemps} \quad (4.3)$$

As described, in section 2.2.2, the scoping rules of PHP are very simple and can be expressed with a global and a local scope. The global scope is necessary, because global variables can always be accessed from a function using the **global** statement. Furthermore the super-global variables resides in the global scope but can always be accessed directly. Only two scopes is enough, because any other variables has to be passed to a function as an argument. This is even the case for anonymous functions, where however the **use** keyword can be used to pass variables to a function when defining the function.

$$\text{Locals} = \text{Globals} = \text{Scope} = \text{Var} \rightarrow \mathcal{P}(\text{HLoc}) \quad (4.4)$$

The scopes are defined as maps from variable names, **Var**, to power-set of heap locations $\mathcal{P}(\text{HLoc})$. While PHP supposedly performs deep copies of values on assignments, letting the scope be a map from variable names to values would not facilitate feature of assigning references to and from variables and array entries. This is done using the **&** operator and makes the heap abstraction is necessary. The heap allows values to be used by multiple variables and arrays which enables properly propagation of changes.

$$\text{Heap} = \text{HLoc} \rightarrow \text{Value} \quad (4.5)$$

The **Temps** and **HeapTemps** are for storing intermediate results. Since these results cannot be referenced there is no need to store them in the heap. By keeping them in a seperate lattice, they can be strongly updated and does not have to, and should not, be passed when switching context, since they are in every respect local to the current context. A single temporary storage mapping from temporary variables to a sum-lattice of values and power-set lattice was considered, this however would involve special handling of \top and \perp elements, which is avoided by this method.

$$\text{Temps} = \text{TVar} \rightarrow \text{Value} \quad (4.6)$$

$$\text{HeapTemps} = \text{THVar} \rightarrow \mathcal{P}(\text{HLoc}) \quad (4.7)$$

The necessity for the latter lattice follows from the fact that reference assignments may be nested, which requires intermediate results shared between nodes

in the control-flow graph. The sets of temporary variables, TVar and THVar , are both finite, following from the control-flow-graph being finite.

The heap locations are allocation site abstractions wrt. context, node, and a natural number. The natural number allows the creation of multiple location per node, which is necessary in e.g. *call*-nodes. Adding the natural number as a factor makes the set of allocation sites possibly infinite, in practise however the set is finite.

$$\text{HLoc} = \Delta \times \mathcal{N} \times \mathbb{N} \quad (4.8)$$

Where \mathcal{N} is the set of nodes in the control flow graph.

An abstract value is defined a product lattice with five factors defined by the Hasse diagrams shown in figure 4.1. These lattices was chosen with the hope of better coercion between values, but others might be considered. E.g. by focusing more on coercion from strings to array indices.

$$\text{Value} = \text{Array} \times \text{String} \times \text{Number} \times \text{Boolean} \times \text{Null} \quad (4.9)$$

Following the results of the dynamic analysis in 3, the array is considered either a set of locations or a map from indices to sets of types. I.e. lists or maps. The sum-lattice has been chosen as opposed to a product-lattice, because the dynamic analysis indicated that arrays seldom changes from lists to maps or vice versa. Top array elements is then a predictor for suspicious behavior. Furthermore the array lattice has an element for the empty array which can become either a list or a map.

$$\text{ArrayList} = \mathcal{P}(L) \quad (4.10)$$

$$\text{ArrayMap} = \text{Index} \rightarrow \mathcal{P}(L) \quad (4.11)$$

The indices of the map-array is an infinite lattice yielding a possibly infinite array-map. Assuming that an infinite sized array existed, this would require an infinite number of writes to a map, which in turn would require an infinite sized program, a recursive function, or a loop. Because of the infinite program is not being possible one of the latter cases must hold. Assuming the cause is a recursive function, then because of a finite number of contexts and allocation sites abstracting the indices is bound to happen and cannot cause an array of infinite size. Now assuming that the array is caused by a loop, then the indices must be generated from previous iterations, meaning that they are generated from information stored on the heap. With a finite number of heap locations and no strong heap update, the indices must be abstracted, thus not yielding an array of infinite size.

$$\text{Index} = \text{String} + \text{Integer} \quad (4.12)$$

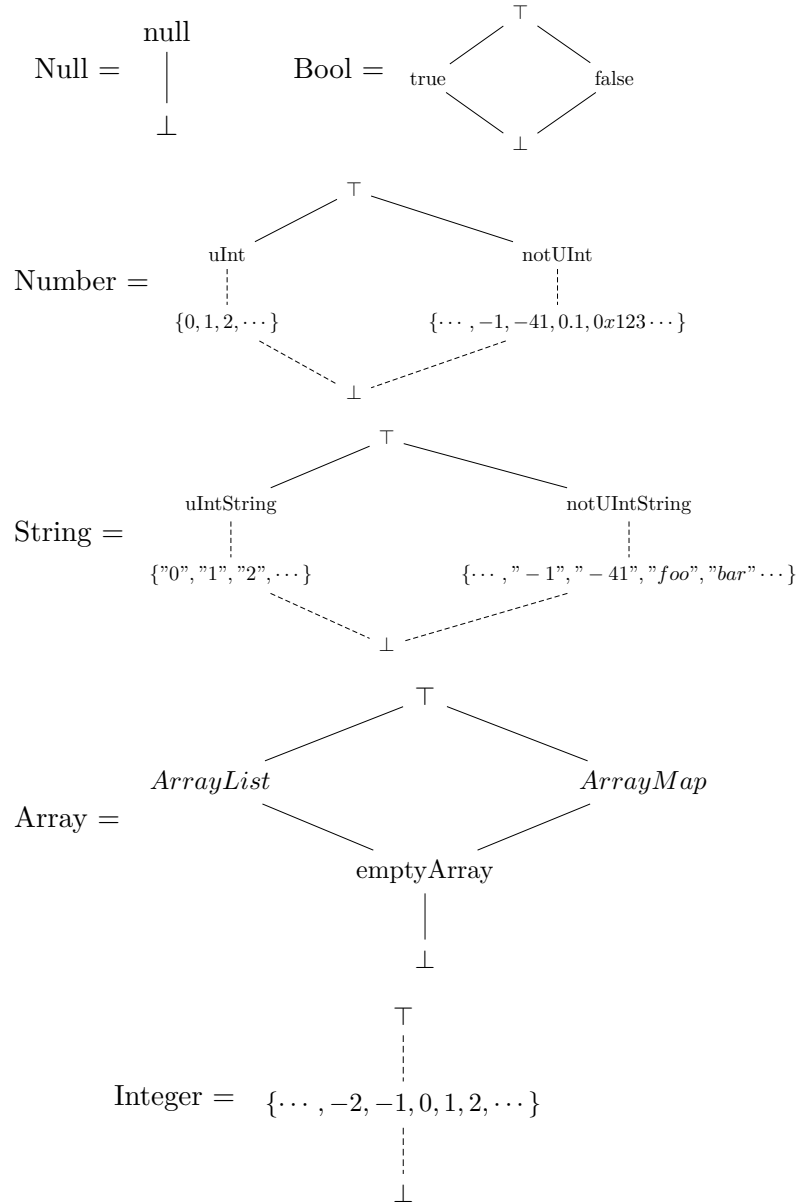


Diagram 4.1: Hasse diagrams of lattices

4.4 Transfer functions

Each node in the control flow graph has a corresponding transfer function. Most of the transfer functions are defined on `State` instead of `AnalysisLattice` i.e they can be defined as $f_{n,\delta} : \text{State} \rightarrow \text{State}$ rather than $f_{n,\delta} : \text{AnalysisLattice} \rightarrow \text{AnalysisLattice}$. This eases the notation and given a state-transfer-function, $f'_{n,\delta}$, the corresponding lattice-transfer function, $f_{n,\delta}$, can be defined as

$$f_{n,\delta}(l) = l[\delta \mapsto f'_{n,\delta}(l(\delta))] \quad (4.13)$$

Where $n \in \mathcal{N}$ is a node in the control-flow graph and $\delta \in \Delta$ is the current context. The transfer functions are defined below.

4.4.1 Operations

Let $n = \text{bop}_{\oplus}(t_l, t_r, t_{tar})$ or $n = \text{sop}_{\oplus}(t_l, t_r, t_{tar})$ then

$$f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = (s_l, s_g, s_h, s_t[t_{tar} \mapsto s(t_l) \oplus s(t_r)], s_{ht}) \quad (4.14)$$

or $n = \text{uop}_{\circ}(t_{val}, t_{tar})$ then

$$f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = (s_l, s_g, s_h, s_t[t_{tar} \mapsto \circ s(t_{val})], s_{ht}) \quad (4.15)$$

The soundness of the binary, unary, and short-circuit operations follows from the subsequent implementation of the abstract evaluation. This is covered, in detail, in section 4.6. These operations solely operates on the temporary variables which acts as intermediate storage for the result of a computation. By not storing these in the heap every update is a strong update, which increases precision.

Since the increment and decrement operations have to read a set of possible locations and update the value of the locations, these are not performed on the temporary variables. Operations on the heap can never be performed by strong update, hence the new values must be joined with the old. The $\text{updateLocations} : \mathcal{P}(\text{HLoc}) \times \text{Heap} \times (\text{HLoc} \rightarrow \text{Value}) \rightarrow \text{Heap}$ function writes a value to the heap using weak updates.

$$\text{updateLocations}(L, h, v) = h[\forall l \in L. l \mapsto h(l) \sqcup v] \quad (4.16)$$

For $n = inc_o(h_{val}, t_{tar})$ the transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{match } \circ \\
& \text{with PreIncrement:} \\
& \text{with PreDecrement:} \\
& \text{let} \\
& \quad s'_h = \text{updateLocations}(\\
& \quad \quad s_{ht}(h_{val}), s_h, l \rightarrow \circ s_h(l)) \\
& \text{in} \\
& \quad (s_l, s_g, s'_h, s_t[t_{tar} \mapsto s'_h(s_{ht}(h_{val}))], s_{ht}) \\
& \text{with PostIncrement:} \\
& \text{with PostDecrement:} \\
& \quad (s_l, s_g, \\
& \quad \quad \text{updateLocations}(s_{ht}(h_{val}), s_h, l \rightarrow \circ s_h(l)), \\
& \quad \quad s_t[t_{tar} \mapsto s_h(s_{ht}(h_{val}))], s_{ht}) \tag{4.17}
\end{aligned}$$

Which updates the heap and the target temporary variable, t_{tar} .

4.4.2 Variables

When writing to a variable, as in the previous section, strong can never occur. The reason for this follows from how PHP performs deep-copy and is covered later in this chapter. The $\text{writeVar} : \text{Var} \times \text{Scope} \times \text{Heap} \times \text{Value} \rightarrow \text{Scope} \times \text{Heap}$ function writes to the heap while ensuring that the provided scope is updated accordingly.

$$\begin{aligned}
\text{writeVar}_{n,\delta}(v, s, h, v_{val}) = & \text{if } s(v) = \emptyset \text{ then} \\
& (s[v \mapsto \{\text{HLoc}(n, \delta, 0)\}], h[\text{HLoc}(n, \delta, 0) \mapsto v]) \\
& \text{else} \\
& (s, \text{updateLocations}(s(v), h, v_{val})) \tag{4.18}
\end{aligned}$$

With a separate Locals and Globals scope, the current scope, wrt. a variable, v , is decided by comparing the current context and the variable name. If the context is empty or if the variable is a super-global, then the current scope is the global scope, else it is the local scope. Deciding whether a variable is a super global, is done by the relation isSuperGlobal which holds if and only if v is either `$_GET`, `$_POST`, `$_SESSION`, `$_COOKIE`, `$_SERVER`, `$_REQUEST`, `$_FILES`, `$_ENV`, or `$GLOBALS`.

For instance in program 4.3, since the variable `$a`, on line 4, is in a function body, the current scope is the local scope, since the variable `$GLOBALS` on line 5 is a super-global, it is in the global scope, and since the variable `$c` is not in a function body, it is in the global scope. The variable `$b`, on line 3 is also in the local scope. This follows from the fact the variable is only an alias for the corresponding global variable, e.g. it shares the same locations as the global value, but if a reference assignment is performed on `$b`, e.g. `$b = &$a`,

it will only effect the local variable, as opposed to a reference assignment to a super-global.

Program 4.3 Scopes

```

1 <?php
2 function f($a){
3     global $b;
4     var_dump($a);
5     var_dump($GLOBALS);
6 }
7 var_dump($c);

```

For $n = \text{varWrite}(v, t_{val}, t_{tar})$ the transfer function is defined as

$$\begin{aligned}
 f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{if } \delta = \Lambda \vee \text{isSuperGlobal}(v) \text{ then} \\
 & \text{let } (g, h) = \text{writeVar}_{n,\delta}(v, s_g, s_h, s_t(t_{val})) \text{ in} \\
 & (s_l, g, h, s_t[t_{tar} \mapsto s_t(t_{val})], s_{ht}) \\
 & \text{else} \\
 & \text{let } (l, h) = \text{writeVar}_{n,\delta}(v, s_l, s_h, s_t(t_{val})) \text{ in} \\
 & (l, s_g, h, s_t[t_{tar} \mapsto s_t(t_{val})], s_{ht}) \quad (4.19)
 \end{aligned}$$

and for $n = \text{varWrite}(v, h_{val}, t_{tar})$ the transfer function is defined as

$$\begin{aligned}
 f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{if } \delta = \Lambda \vee \text{isSuperGlobal}(v) \text{ then} \\
 & (s_l, s_g[v \mapsto s_{ht}(h_{val})], s_h, s_t[t_{tar} \mapsto s_h(s_{ht}(h_{val}))], s_{ht}) \\
 & \text{else} \\
 & (s_l[v \mapsto s_{ht}(h_{val})], s_g, s_h, s_t[t_{tar} \mapsto s_h(s_{ht}(h_{val}))], s_{ht}) \quad (4.20)
 \end{aligned}$$

In the latter case the variable is always strongly updated. This is sound because the current language subset of PHP offers no ambiguity with regards to which variable currently being updated. Specifically because the infamous variable-variable feature has been omitted.

Besides resolving the scope, as above, reading a variable is quite straight forward. In order to be sound however, the transfer function does need to take uninitialized variables into account. When reading an uninitialized variable in PHP, the default value is **NULL**, therefor if reading to a temporary variable, the results should be $\text{Value}(\text{Null}(\top))$ if a variable, in the current scope, is not pointing to any locations. Else the result should be the joined value of the heap

locations. For $n = \text{varRead}(v, t_{tar})$ the transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } s = \\
& \text{if } \delta = \Lambda \vee \text{isSuperGlobal}(v) \text{ then } s_g \text{ else } s_l \\
& \text{in} \\
& \text{let } v = \\
& \text{if } s(v) = \emptyset \text{ then Value(Null}(\top)) \text{ else } s_h(s(v)) \\
& \text{in} \\
& (s_l, s_g, s_h, s_t[t_{tar} \mapsto v], s_{ht})
\end{aligned} \tag{4.21}$$

When reading the locations of a variable, special care has to be shown when the variable is uninitialized. Since reading the same variable twice, with no intermediate modification, must return the same locations, the variable has to be initialized when first read. This is done by the $\text{initializeVariable}_{n,\delta} : \text{Var} \times \text{Scope} \rightarrow \text{Scope}$ function, which creates a new location in the provided scope, if none exists.

$$\text{initializeVariable}_{n,\delta}(v, s) = \text{if } s(v) = \emptyset \text{ then } s[v \mapsto \{\text{HLoc}(n, \delta, 0)\}] \text{ else } s \tag{4.22}$$

For $n = \text{varRead}(v, h_{tar})$ the transfer function becomes

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{if } \delta = \Lambda \vee \text{isSuperGlobal}(v) \text{ then} \\
& \text{let } s'_g = \text{initializeVariable}_{n,\delta}(v, s_g) \text{ in} \\
& (s_l, s'_g, s_h, s_t, s_{ht}[h_{tar} \mapsto s'_g(v)]) \\
& \text{else} \\
& \text{let } s'_l = \text{initializeVariable}_{n,\delta}(v, s_l) \text{ in} \\
& (s'_l, s_g, s_h, s_t, s_{ht}[h_{tar} \mapsto s'_l(v)])
\end{aligned} \tag{4.23}$$

4.4.3 Arrays

There are four types of array operations; initialize, read, write and append, with one, two, three and four different signatures respectively. For $n = \text{arrayInit}(t_{tar})$ the transfer function becomes

$$f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = (s_l, s_g, s_h, s_t[t_{tar} \mapsto \text{Value(emptyArray)}], s_{ht}) \tag{4.24}$$

which is trivially sound, since all it does is to initialize an empty array in the given temporary variable.

Array initialization

Array initialization happens when writing the expression $[\dots]$ or $\text{array}(\dots)$, where the dots indicate initial content. Remember however that arrays need not be initialized explicitly. An array write or append to an uninitialized variable

creates an array and writes to it. This does not yield a *arrayInit* node in the CFG. The initial content is added to the array immediately after initialization by the following *arrayAppend*(t_{val}, t_{ar}) node and the *arrayWrite*(t_{val}, t_{key}, t_{ar}) node.

Array append

Appending a value, stored in the temporary values, to an array likewise stored in the temporary values, is performed by first storing the value in the heap, at some location, l . Thereafter the array is joined with an list-array of the set containing only l . While this implies that an append operation on a map-array results in the \top array, thus loosing all precision, our hypothesis states that the append operation should only be performed on lists. Therefore this loss should not occur in a *good* program. For $n = \text{arrayAppend}(t_{val}, t_{ar})$ the transfer function becomes

$$\begin{aligned}
 f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let} \\
 & (v_a, v_s, v_n, v_b, v_u) = s(t_{ar}), \\
 & l = \text{HLoc}(\delta, n, 0) \\
 & \text{in} \\
 & (s_l, s_g, \\
 & \quad s_h[l \mapsto s_t(t_{val})], \\
 & \quad s_t[t_{ar} \mapsto (v_a \sqcup \text{ArrayList}(\{l\}), v_s, v_n, v_b, v_u)], \\
 & \quad s_{ht}) \tag{4.25}
 \end{aligned}$$

As mentioned earlier, this append node only occurs immediately after array initialization as a method for inserting initial content into an array. E.g. for the expression `[1,2,3]` the numbers, 1, 2, and 3 are appended to the array initialized by the brackets.

Probably the most common use of array appends are in a normal assignment, e.g. `$a[] = 42;` or `$a['someKey'][] = 42;`. In this case the value, 42, is stored in temporary storage, and the location of the array being modified is stored in the heap. The operation then becomes; appending a value on a set of locations, which is done in the same manner as the previous node. By for each possible array location joining the existing array with a list-array. Here however the value being appended is also added to the temporary variable t_{tar} , since this is what an append returns in PHP, e.g. `echo $a[] = 42;` prints the number 42.

For $n = \text{arrayAppend}(h_{var}, t_{val}, t_{tar})$ the transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } l' = \text{HLoc}(\delta, n, 0) \text{ in} \\
& \text{let } s'_t = s_t[t_{tar} \mapsto s_t(t_{val})] \text{ in} \\
& \text{let } s'_h = s_h[\\
& \quad l' \mapsto s_t(t_{val}), \\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \text{let } (v_a, v_s, v_n, v_b, v_u) = s_h(l) \text{ in} \\
& \quad l \mapsto (v_a \sqcup \text{ArrayList}(\{l'\}), v_s, v_n, v_b, v_u)] \\
& \text{in} \\
& (s_l, s_g, s'_h, s'_t, s_{ht})
\end{aligned} \tag{4.26}$$

Another type of append is in the context of a reference assignment, e.g. $\$a[] = \&\b . Here the locations pointed to by $\$b$, should be appended to the array(s) at $\$a$. I.e appending a value, in the form of a set of locations to a set of locations. This is done like before, the only difference being that the list-array of which the existing values are joined, does not contain a single new location, rather the set of locations corresponding to the value. Again the target variable in temporary storage must be set to the value assigned, not the location-set representing the value.

For $n = \text{arrayAppend}(h_{var}, h_{val}, t_{tar})$ the transfer function becomes

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } s'_t = s_t[t_{tar} \mapsto s_h(s_{ht}(h_{val}))] \text{ in} \\
& \text{let } s'_h = s_h[\\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \text{let } (v_a, v_s, v_n, v_b, v_u) = s_h(l) \text{ in} \\
& \quad l \mapsto (v_a \sqcup \text{ArrayList}(s_{ht}(h_{val})), v_s, v_n, v_b, v_u)] \\
& \text{in} \\
& (s_l, s_g, s'_h, s'_t, s_{ht})
\end{aligned} \tag{4.27}$$

The final array-append operation occurs when an array is appended and immediately thereafter accessed, e.g. $\$a[][\text{'key'}]$ where a location is appended to the array at $\$a$ and immediately thereafter implicitly initialized and written to. As previously mentioned an array does not have to be explicitly initialized. The location, l' , is created and appended in the same way as previously to all possible array locations. Since the heap lattice element initializes new locations to $\text{Value}(\text{Null}(\top))$ it is important, and sound, to set l' to $\text{Value}(\perp)$ since the location will be joined with another array immediately after.

For $n = \text{arrayAppend}(h_{var}, h_{tar})$ the transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } l' = \text{HLoc}(\delta, n, 0) \text{ in} \\
& \text{let } s'_{ht} = s_{ht}[h_{tar} \mapsto \{l'\}] \text{ in} \\
& \text{let } s'_h = s_h[\\
& \quad l' \mapsto \text{Value}(\perp), \\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \quad \text{let } (v_a, v_s, v_n, v_b, v_u) = s_h(l) \text{ in} \\
& \quad l \mapsto (v_a \sqcup \text{ArrayList}(\{l'\}), v_s, v_n, v_b, v_u)] \\
& \text{in} \\
& (s_l, s_g, s'_h, s_t, s'_{ht})
\end{aligned} \tag{4.28}$$

Array read and write

When writing or reading from an array, given a value, v , as key, the value must first be coerced to an array index. The easy approach would be to use the coercion function directly $c_{\text{Value}, \text{ArrayIndex}}(v)$, introduced in the section 4.5, which coerces and then joins all factors. This approach would for instance coerce the value $v = (\perp, \text{"foo"}, 4, \perp, \perp)$ to the index $i = c_{\text{Array}, \text{ArrayIndex}}(\perp) \sqcup \dots \sqcup c_{\text{Null}, \text{ArrayIndex}}(\perp) = \text{"foo"} \sqcup 4 = \top$.

Another approach would be to consider a set of possible indices, I , rather than a single index, by coercing the factors individually and access the array with each index. For the previous example v would yield the set $I = \{\perp, \text{"foo"}, 4\}$. Since the value will most likely contain at least one \perp factor, the set of indices will also contain the \perp array index, which will become a problem.

The problem becomes apparent when first considering how values written to and read from map-arrays. Writing index i with location set L on some map-array, a , should ideally be done by joining the location set of each entry in a with L where i is contained in the corresponding key. E.g. $a[\forall d \in \text{dom}(a) \wedge i \sqsubseteq d.d \mapsto a(d) \sqcup L]$. Updating a possibly infinite domain could be done lazily, but deciding containment of two lattices, where one has infinitely many changes, is not practically feasible. As a compromise the only entry updated in a is key i with the joined set $a(i) \sqcup L$. This compromise entails that when reading i' from map-array a , the set of possible keys is all $d \in \text{dom}(a)$ where $d \sqsubseteq i'$ or $i' \sqsubseteq d$, which is practically feasible.

Returning to the problematic \perp factors. Since most writes would contain a at least one \perp factor, most writes would, with the second approach, write to the \perp index, and since \perp is contained in all indices, massive loss of precision is ensured. Therefore a third option is to only consider coerced factors, of the key value, that are not contained in other factors. This reduces the set of the previous example to $I = \{\text{"foo"}, 4\}$. These are the indices returned by the

function *indices* : Value \rightarrow ArrayIndex*

$$\begin{aligned}
indices(v) = & \text{let } (v_a, v_s, v_n, v_b, v_u) = v \text{ in} \\
& \text{let } I = \{c_{\text{Array}, \text{Index}}(v_a), \\
& \quad c_{\text{String}, \text{Index}}(v_s), \\
& \quad c_{\text{Number}, \text{Index}}(v_n), \\
& \quad c_{\text{Boolean}, \text{Index}}(v_b), \\
& \quad c_{\text{Null}, \text{Index}}(v_u)\} \text{ in} \\
& I \setminus \{j | i, j \in I \wedge j \sqsubseteq i \wedge i \neq j\}
\end{aligned} \tag{4.29}$$

Using the above function, we are able to generalize reading from an array as a function $readArray : \text{Value} \times \text{Value} \times \text{Heap} \rightarrow \mathcal{P}(L)$ which given a value, key, and heap returns a set of possible value locations. Reading an index from a list, returns the set of locations in the list, since no key information is kept. Reading from \perp or `emptyMap` results in the empty set, since they contain no locations. Finally reading from \top results in all possible locations, e.g. the top element of $\mathcal{P}(L)$, since an array can potentially point to every location in the heap, including itself.

$$\begin{aligned}
readArray(v, k, h) = & \text{let } (v_a, v_s, v_n, v_b, v_u) = v \text{ in} \\
& \text{match } v_a \\
& \quad \text{with } \top : dom(h) \\
& \quad \text{with } ArrayList(L) : L \\
& \quad \text{with } ArrayMap(m) : \\
& \quad \quad \bigcup_{d \in dom(m) \wedge \exists i \in indices(k). i \sqsubseteq d \vee d \sqsubseteq i} m(d) \\
& \quad \text{with } emptyArray : \emptyset \\
& \quad \text{with } \perp : \emptyset
\end{aligned} \tag{4.30}$$

In the same manner can the act of writing to an array be generalized to the function $writeArray : \text{Value} \times \text{Value} \times \mathcal{P}(L) \rightarrow \text{Value}$. Here writing to a \top array results in a \top array, writing to a map, updates the keys as discussed before, and writing to anything else either returns a list or a map depending on the type of keys being used. If some of the indices are strings then a map is

returned, else a list is returned.

$$\begin{aligned}
writeArray(v, k, L) = & \text{let } (v_a, v_s, v_n, v_b, v_u) = v \text{ in} \\
& \text{let } m = \text{ArrayMap}([\forall i \in indices(k).i \mapsto L]) \text{ in} \\
& (\text{match } v_a \\
& \quad \text{with } \text{ArrayList}(L'): \\
& \quad \quad \text{if } \exists i \in indices(k).i \text{ is String}(_) \text{ then} \\
& \quad \quad \quad \text{ArrayMap}([\top \mapsto L']) \sqcup m \text{ else} \\
& \quad \quad \quad \text{ArrayList}(L \cup L') \\
& \quad \text{with } \text{ArrayMap}(m'): m' \sqcup m \\
& \quad \text{with } \text{emptyArray}: \\
& \quad \quad \text{if } \exists i \in indices(k).i \text{ is String}(_) \text{ then} \\
& \quad \quad \quad m \text{ else } \text{ArrayList}(L) \\
& \quad \text{with } \perp: \\
& \quad \quad \text{if } \exists i \in indices(k).i \text{ is String}(_) \text{ then} \\
& \quad \quad \quad m \text{ else } \text{ArrayList}(L) \\
& \text{with } \top: \top, v_s, v_n, v_b, v_u)
\end{aligned} \tag{4.31}$$

With these functions in place, the transfer functions for the array-read and write nodes can be defined. For reading from an array in temporary storage, e.g. right side of the assignment $\$b = \$a['key']$, the node is $n = arrayRead(t_{ar}, t_{key}, t_{tar})$ where the value read from the array should be stored at t_{tar} . The query is always joined with $\text{Value}(\text{Null}(\top))$ since an entry might not be set. The transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } L = readArray(s_t(t_{ar}), s_t(t_{key}), s_h) \text{ in} \\
& \text{let } v = \text{Value}(\text{Null}(\top)) \sqcup s_h(L) \text{ in} \\
& (s_l, s_g, s_h, s_t[t_{tar} \mapsto v], s_{ht})
\end{aligned} \tag{4.32}$$

Another array-read operation is when resolving the locations of an entry corresponding to a given key, e.g. resolving entry $\$a['foo']$ in the assignment $\$a['foo'][] = 42$. This is done by joining the locations of each possible entry, adding a new location to each entry yielding an empty location-set. Updating empty entries ensures that subsequent modifications of the returned location-set is propagated to every possible array. For this operation the node is $n =$

$arrayRead(h_{var}, t_{key}, h_{tar})$ and transfer function becomes

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } l = \text{HLoc}(\delta, n, 0) \text{ in} \\
& \text{let } s'_{ht} = s_{ht}[h_{tar} \mapsto \\
& \quad \cup_{l' \in s_{ht}(h_{var})} cardCheck(\\
& \quad \quad readArray(s_h(l'), s_t(t_{key}), s_h), l)] \\
& \text{in} \\
& \text{let } s'_h = \\
& \quad s_h[\forall l' \in s_{ht}(h_{var}) \\
& \quad \quad \wedge readArray(s_h(l'), s_t(t_{key}), s_h) = \emptyset. \\
& \quad \quad l' \mapsto writeArray(s_h(l'), s_t(t_{key}), \{l\})] \\
& \text{in} \\
& (s_l, s_g, s'_h, s_t, s'_{ht})
\end{aligned} \tag{4.33}$$

where

$$cardCheck(L, l) = \text{if } L = \emptyset \text{ then } \{l\} \text{ else } L \tag{4.34}$$

As mentioned earlier in this section, immediately after an explicit array initialization follows writing and appending the initial content of the array. E.g. for the array creation `['a'=>1, 'b'=>2]`, follows two array writes, one with value 1 at key 'a' and with value 2 at key 'b', respectively. These writes are expressed with $n = arrayWrite(t_{key}, t_{val}, t_{ar})$ and are performed using the *writeArray* function. Here the value, at temporary variable t_{val} is stored in the heap at a new location, which is consequently written to the existing array. The transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } l' = \text{HLoc}(\delta, n, 0) \text{ in} \\
& \text{let } v' = \text{in} \\
& \quad (s_l, s_g, s_h[l' \mapsto s_t(t_{val})], \\
& \quad \quad s_t[t_{ar} \mapsto writeArray(s_t(t_{ar}), s_t(t_{key}), \{l'\})], s_{ht})
\end{aligned} \tag{4.35}$$

When the array is residing in the heap, rather than temporary storage, e.g. `$a['key'] = 42;`, resolving the possible locations of the array must first be performed, thereafter the write operation can be performed. This is expressed with the node $n = arrayWrite(h_{var}, t_{key}, t_{val}, t_{tar})$ where the transfer function

is

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } l' = \text{HLoc}(\delta, n, 0) \text{ in} \\
& \text{let } s'_t = s_t[t_{tar} \mapsto s_t(t_{val})] \text{ in} \\
& \text{let } s'_h = s_h[\\
& \quad \forall l \in s_h(h_{var}). \\
& \quad \quad \forall l'' \in \text{readArray}(s_h(l), s_t(t_{key}), s_h). \\
& \quad \quad \quad l'' \mapsto s_h(l'') \sqcup s_t(t_{val})] \text{ in} \\
& \text{let } s''_h = s'_h[\\
& \quad l' \mapsto s_t(t_{val}), \\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \quad \quad l \mapsto \text{writeArray}(s'_h(l), s_t(t_{key}), \{l'\})] \\
& \text{in} \\
& (s_l, s_g, s''_h, s'_t, s_{ht})
\end{aligned} \tag{4.36}$$

Writing a value to an array can be viewed as two cases; writing to an existing entry and writing to a new entry. The first case entails that the value, stored in the heap, should be updated and the other that a new location, pointing to the new value, should be added to the entry. In order to perform a sound write, both cases are performed.

Writing to an array in the context of a reference assignment, e.g. `$a['key'] = &$b;`, is performed by writing the set of locations pointed to by `$b` to the entry at `'key'` in array `$a`. This is expressed with the node $n = \text{arrayWrite}(h_{var}, t_{key}, h_{val}, t_{tar})$ where the transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{let } s'_t = s_t[t_{tar} \mapsto s_t(s_h(s_{ht}(h_{val})))] \text{ in} \\
& \text{let } s'_h = s_h[\\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \quad \quad l \mapsto \text{writeArray}(s_h(l), s_t(t_{key}), s_{ht}(h_{val}))] \\
& \text{in} \\
& (s_l, s_g, s'_h, s'_t, s_{ht})
\end{aligned} \tag{4.37}$$

Notice that the this operation does not update the heap values as the previous functions. This is sound because assigning new locations are in practice overwriting old ones. If the array keys were to be exactly resolved a strong update could be performed here, just as the case with the reference assignment to variables.

The array operations might seem similar to the operations on variables just with another level of ambiguity. Viewing the scopes as map-arrays from strings to location-sets is not far from how PHP implements scopes and may provide a good intuition as to how and why the variable-variable feature is implemented.

Notice that no strong updates are performed on heap values This is in order to maintain soundness and follows from how PHP performs deep copy, which is described in section 2.2.3. By copying the references PHP opens the possibility

for the modification of a deep-copied array through another variable/array, with no reference assignment from the array. In order to be sound, the analysis has to assume that no arrays are deep copied, but instead shares the internal references of the original array. From this follows that no strong-updates can be performed on any location, because it might result in an update of an array which in practice never share the location of the variable. This issue is illustrated in program 4.4. If strong updates were allowed, updating **\$c** in the last line would result in **\$a** rightfully and **\$b** wrongfully being updated to the list containing the number two, **[2]**, since the two arrays share the same internal locations. By only performing weak updates, the two arrays becomes a list of `UIntNumber` which is sound.

Program 4.4

```

1 $a = [1];
2 $b = $a;
3 $c = &$a[0];
4 $c = 2;

```

4.4.4 Function calls

The transfer functions for function call, *call* and *result* nodes, differs from the other functions. For $n = call_{fn}(c_1, \dots, c_n)$ the transfer function $f_{n,\delta} : \text{AnalysisLattice} \rightarrow \text{AnalysisLattice}$ sets up the local scope for the function body of fn . This scope is initially empty with exception of the function arguments being set by reference or by value, depending on call arguments, c_1, \dots, c_n , being a `THVar` or a `TVar` respectively.

If the argument is passed by reference, then the corresponding argument is set, in the local scope, to point at the provided heap locations. If the argument is passed by value, then the argument is pointing to a newly created heap location, which in turns points to the value. The second case is one of the reasons for `HLoc` being defined as a product of context, node, and a natural number. Without the third factor all, value-passed, arguments would be written to the same heap-location. This is avoided by setting the number in the heap location to the position of the argument in question.

The global scope and heap is preserved in the new scope, while the tempo-

rary maps both are *emptied*. The transfer function is defined as

$$\begin{aligned}
f_{n,\delta}(l) = & \text{let } \delta' = \text{addCallNode}(\delta, n) \text{ in} \\
& \text{let } (_, s_g, s_h, s_t, _) = l[\delta] \text{ in} \\
& \text{let } s_l = [\forall (v, i) \in \text{args}(fn). v \mapsto \\
& \quad \text{match } c_i \\
& \quad \quad \text{with TVar: } \{\text{HLoc}(\delta, \text{startNode}(fn), i)\} \\
& \quad \quad \text{with THVar: } s_h(c_i)] \\
& \text{in} \\
& \text{let } s'_h = [\forall (_, i) \in \text{args}(fn). \text{HLoc}(\delta, \text{startNode}(fn), i) \mapsto \\
& \quad \text{match } c_i \\
& \quad \quad \text{with TVar: } s_t(c_i) \\
& \quad \quad \text{with THVar: } s_h(\text{HLoc}(\delta, \text{startNode}(fn), i))] \\
& \text{in} \\
& (s_l, s_g, s'_h, [], [])
\end{aligned} \tag{4.38}$$

where the $\text{addCallNode} : \Delta \times \text{CallNode} \rightarrow \Delta$ is deciding the target context from the current, the $\text{args} : \text{FunctionNames} \rightarrow (\text{Var} \times \mathbb{N})^*$ function, given a function name, returns a list of arguments expressed as pairs of variable names and positions, and $\text{startNode} : \text{FunctionNames} \rightarrow \text{StartNode}$, given a function name, returns the unique *start* node of that function. The heap locations created are associated with the start node, rather than the call node, because of efficiency.

After running a function it is the task of the transfer function of the result node, to restore the old execution context. For $n = \text{result}_{\text{call}_{fn}}(_)$ the transfer functions are defined as functions from two lattices to a single lattice: $f_{n,\delta_{\text{call}},\delta_{\text{exit}}} : \text{AnalysisLattice} \times \text{AnalysisLattice} \rightarrow \text{AnalysisLattice}$ where the first lattice is the lattice passed to the transfer function of the call node, call_{fn} and δ_{call} is the original context. Depending on the argument of the *result* node the transfer function is defined as either

$$\begin{aligned}
f_{n,\delta_{\text{call}},\delta_{\text{exit}}}(l_{\text{call}}, l_{\text{exit}}) = & \text{let } \text{exit}(c_1, \dots, c_n) = \text{exitNode}(fn) \text{ in} \\
& \text{let } (s_l, _, _, s_t, s_{ht}) = l_{\text{call}}(\delta_{\text{call}}) \text{ in} \\
& \text{let } (_, s_g, s_h, s'_t, s'_{ht}) = l_{\text{exit}}(\delta_{\text{exit}}) \text{ in} \\
& \text{let } v = \\
& \quad \bigsqcup_{0 < i \leq n} \text{match } c_i \\
& \quad \quad \text{with TVar: } s_t(c_i) \\
& \quad \quad \text{with THVar: } s_h(s_{ht}(c_i)) \\
& \text{in} \\
& (s_l, s_g, s_h, s_t[t_{\text{val}} \mapsto v], s_{ht})
\end{aligned} \tag{4.39}$$

for $n = \text{result}_{\text{call}_{f_n}}(t_{\text{val}})$ or for $n = \text{result}_{\text{call}_{f_n}}(h_{\text{val}})$ as

$$\begin{aligned}
f_{n, \delta_{\text{call}}, \delta_{\text{exit}}}(l_{\text{call}}, l_{\text{exit}}) = & \text{let } \text{exit}(c_1, \dots, c_n) = \text{exitNode}(fn) \text{ in} \\
& \text{let } (s_l, _, _, s_t, s_{ht}) = l_{\text{call}}(\delta_{\text{call}}) \text{ in} \\
& \text{let } (_, s_g, s_h, s'_t, s'_{ht}) = l_{\text{exit}}(\delta_{\text{exit}}) \text{ in} \\
& \text{let } L = \\
& \quad \bigcup_{0 < i \leq n} \text{match } c_i \\
& \quad \quad \text{with } TVar: \{\text{HLoc}(n, \delta_{\text{call}}, i)\} \\
& \quad \quad \text{with } THVar: s_{ht}(c_i) \\
& \text{in} \\
& \text{let } s'_h = s_h[\forall 0 < i \leq n. \text{HLoc}(n, \delta_{\text{call}}, i) \mapsto \\
& \quad \text{match } c_i \\
& \quad \quad \text{with } TVar: s_t(c_i) \\
& \quad \quad \text{with } THVar: s_h(\text{HLoc}(n, \delta_{\text{call}}, i))] \\
& \text{in} \\
& l_{\text{call}}[\delta_{\text{call}} \mapsto (s_l, s_g, s'_h, s_t, s_{ht}[h_{\text{val}} \mapsto L])] \quad (4.40)
\end{aligned}$$

where the $\text{exitNode} : \text{FunctionName} \rightarrow \text{ExitNode}$ function, given a function name, returns the corresponding unique exit node. These functions will return an lattice containing all local values, temps and local scope, from the call-context and the global values, global scope and heap, from the exit-context. The possible result of the function-call is gathered from the exit-node and saved in either temporary- or heap-temporary-variables, depending on the function being pass by value or by reference respectively. In the latter case the position of the *exit-argument* is again used to decide which heap locations to save the value, if the function returns a value rather than references. Since the number of arguments in any function definition, the number of return statements in any function body, and the number of initialized arrays in the start lattice in practice is finite, so is the number of heap-locations.

4.4.5 Other transfer functions

Two interesting transfer functions remains. The function for $n = \text{constRead}(c, t_{\text{tar}})$, which converts a constant c to a lattice using the $\text{value} : \mathcal{C} \rightarrow \text{Value}$ function, which works as one would expect, is defined as

$$f_{n, \delta}((s_l, s_g, s_h, s_t, s_{ht})) = (s_l, s_g, s_h, s_t[t_{\text{tar}} \mapsto \text{value}(c)], s_{ht}) \quad (4.41)$$

and finally for $n = \text{global}(v_0, v_1, \dots, v_{n-1})$; which creates variables in the local scope, sharing the locations of the corresponding variable in the global scope. If the variables points to no locations, a new location must be added to the global and local scope. If the current scope is empty, then no modifications are

made to the input lattice. The transfer function is defined as

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \text{match } \delta \\
& \text{with } \Lambda: (s_l, s_g, s_h, s_t, s_{ht}) \\
& \text{with } _: \\
& \quad \text{let } s'_g = s_g[\forall 0 \leq i \leq n. v_i \mapsto \\
& \quad \quad \text{if } s_g[v_i] = \emptyset \text{ then} \\
& \quad \quad \quad \{\text{HLoc}(n, \delta, i)\} \text{ else } s_g[v_i]] \\
& \quad \text{in} \\
& \quad \text{let } s'_l = s_l[\forall 0 \leq i \leq n. v_i \mapsto s'_g(v_i)] \text{ in} \\
& \quad (s'_l, s'_g, s_h, s_t, s_{ht})
\end{aligned} \tag{4.42}$$

All other transfer functions are the identity function: $f_{n,\delta}(l) = l$.

4.5 Coercion

In order to fully utilize the benefits of a dynamic typed language, PHP supports coercion from most types to scalars (strings, numbers, boolean, etc.). Coercion is primarily used when performing binary-, unary-operators, and when accessing arrays. Coercion to an array index must be treated as a separate case because it behaves differently from string or number coercion. E.g. accessing an array with key `$key`, `$a[$key]`, if `$key = 42` or `$key = "42"` then the entry at integer 42 is accessed, which is similar to string-to-number coercion. The same entry is accessed with `$key = 42.1` or `$key = 4.34E1`, because keys must be either integers or strings. Accessing with `$key = "42.1"` accesses the entry with string key `"42.1"`.

Since there is no language specification, these coercion rules has largely been discovered by manual inspection, using different operators, e.g. number coercion has been explored by adding values of various types together, with the binary `+`-operator, and string coercion with the string-concat `.`-operator.

Coercion on abstract values can be performed using the $c_{\alpha,\beta} : \alpha \rightarrow \beta$ function, where $\alpha \in \{\text{Value}, \text{Array}, \text{Number}, \text{String}, \text{Null}, \text{Boolean}\}$ and $\beta \in \{\text{Number}, \text{Value}, \text{String}, \text{Index}, \text{Boolean}\}$.

Coercing to and from Value's is defined by the following functions

$$c_{\alpha, \text{Value}}(v) = \begin{cases} (v, \perp, \perp, \perp, \perp) & \text{if } \alpha = \text{Array} \\ (\perp, v, \perp, \perp, \perp) & \text{if } \alpha = \text{String} \\ (\perp, \perp, v, \perp, \perp) & \text{if } \alpha = \text{Number} \\ (\perp, \perp, \perp, v, \perp) & \text{if } \alpha = \text{Boolean} \\ (\perp, \perp, \perp, \perp, v) & \text{if } \alpha = \text{Null} \end{cases}$$

$$\begin{aligned}
c_{\text{Value}, \beta}((v_1, v_2, v_3, v_4, v_5)) = & c_{\text{Array}, \beta}(v_1) \sqcup c_{\text{String}, \beta}(v_2) \\
& \sqcup c_{\text{Number}, \beta}(v_3) \sqcup c_{\text{Boolean}, \beta}(v_4) \sqcup c_{\text{Null}, \beta}(v_5)
\end{aligned}$$

where

$$\begin{aligned}
c_{\text{Array},\text{String}}(v) &= \begin{cases} \text{"Array"} & \text{if } v \neq \perp \\ \perp & \text{else} \end{cases} & c_{\text{Null},\text{String}}(v) &= \begin{cases} "" & \text{if } v = \top \\ \perp & \text{else} \end{cases} \\
c_{\text{Number},\text{String}}(v) &= \begin{cases} \text{uIntStr} & \text{if } v = \text{uInt} \\ \text{notUIntStr} & \text{if } v = \text{notUInt} \\ \top & \text{if } v = \top \\ \perp & \text{if } v = \perp \\ \text{string}(v) & \text{else} \end{cases} & c_{\text{Bool},\text{String}}(v) &= \begin{cases} "" & \text{if } v = \text{false} \\ "1" & \text{if } v = \text{true} \\ v & \text{else} \end{cases} \\
c_{\text{Array},\text{Bool}}(v) &= \begin{cases} \text{false} & \text{if } v = \text{emptyArray} \\ \perp & \text{if } v = \perp \\ \top & \text{else} \end{cases} & c_{\text{Null},\text{Bool}}(v) &= \begin{cases} \text{false} & \text{if } v = \top \\ \perp & \text{else} \end{cases} \\
c_{\text{Number},\text{Bool}}(v) &= \begin{cases} \text{false} & \text{if } v = 0 \\ \perp & \text{if } v = \perp \\ \top & \text{if } v = \text{uInt} \vee v = \top \\ \text{true} & \text{else} \end{cases} & c_{\text{String},\text{Bool}}(v) &= \begin{cases} \text{false} & \text{if } v = "" \vee v = \text{"0"} \\ \perp & \text{if } v = \perp \\ \top & \text{if } v = \text{uIntString} \vee v = \top \\ \text{true} & \text{else} \end{cases} \\
c_{\text{Null},\text{Number}}(v) &= \begin{cases} 0 & \text{if } v = \top \\ \perp & \text{else} \end{cases} & c_{\text{Array},\text{Number}}(v) &= \perp \\
c_{\text{Bool},\text{Number}}(v) &= \begin{cases} 1 & \text{if } v = \text{true} \\ 0 & \text{if } v = \text{false} \\ \text{uInt} & \text{if } v = \top \\ \perp & \text{if } v = \perp \end{cases} & c_{\text{String},\text{Number}}(v) &= \begin{cases} \text{num}(v) & \text{if } \text{isNumber}(v) \\ \text{uInt} & \text{if } v = \text{uIntString} \\ \top & \text{if } v = \text{notUIntString} \\ \top & \text{if } v = \top \\ \perp & \text{if } v = \perp \\ 0 & \text{else} \end{cases} \\
c_{\text{Array},\text{Index}}(v) &= \perp & c_{\text{Bool},\text{Index}}(v) &= c_{\text{Bool},\text{Number}}(v) \\
c_{\text{Number},\text{Index}}(v) &= \begin{cases} \text{int}(v) & \text{if } \text{isNumber}(v) \\ \perp & \text{if } v = \perp \\ \top & \text{else} \end{cases} & c_{\text{String},\text{Index}}(v) &= \begin{cases} \text{int}(v) & \text{if } \text{isInteger}(v) \\ v & \text{if } \text{isString}(v) \\ \top & \text{else} \end{cases} \\
c_{\text{Null},\text{Index}}(v) &= \begin{cases} "" & \text{if } v = \top \\ \perp & \text{else} \end{cases} & c_{\alpha,\alpha}(v) &= v
\end{aligned}$$

and *int*, *num*, and *string* are creating an integer, number, and string respectively, in the obvious way. E.g. *int*(42.1) = 42, *string*(1337) = "1337", and *number*("1337") = 1337. Furthermore the predicates *isNumber*, *isInteger*, *isString* holds if and only if the value can be interpreted as a number, integer, or string respectively. E.g. *isNumber*("42.1") holds while *isInteger*("42.1") does not.

A few interesting cases of coercion in PHP include; Any array coerced to a string will result in the string **"Array"**, **null** is string-coerced to the empty string, both the empty string and the string literal **"0"** are boolean-coerced to boolean **false**, boolean **false** is string-coerced to the empty string, and An empty array is boolean-coerced to boolean **false**.

4.6 Abstract evaluation

In order to maintain some precision when evaluating binary- and unary-operators, a table is created for each operation defining the result of the evaluating a given input abstractly. These tables are available in appendix B, with some interesting cases presented below.

Operator	Name	Signature	
<code>x + y</code>	Addition	Number \times Number \rightarrow Number	
<code>x - y</code>	Subtraction		
<code>x * y</code>	Multiplication		
<code>x ** y</code>	Exponentiation		
<code>x / y</code>	Division	Number \times Number \rightarrow Value	
<code>x % y</code>	Modulo		
<code>x == y</code>	Equal	Value \times Value \rightarrow Boolean	
<code>x != y</code>	Not equal		
<code>x === y</code>	Identical		
<code>x !== y</code>	Not identical		
<code>x < y</code>	Less than		
<code>x <= y</code>	Less than or equal		
<code>x > y</code>	Greater than		
<code>x >= y</code>	Greater than or equal		
<code>x && y</code>	Logical and	Boolean \times Boolean \rightarrow Boolean	
<code>x AND y</code>			
<code>x y</code>			Logical or
<code>x OR y</code>			
<code>x XOR y</code>	Exclusive or		
<code>x . y</code>	String concatenation	String \times String \rightarrow String	

Table 4.1: Binary operators

Operator	Name	Signature
<code>! x</code>	Negation	$\text{Boolean} \rightarrow \text{Boolean}$
<code>- - x</code>	Pre-decrement	$\text{Value} \rightarrow \text{Value}$
<code>x - -</code>	Post-decrement	
<code>+ + x</code>	Pre-increment	
<code>x + +</code>	Post-increment	
<code>- x</code>	Unary Minus	

Table 4.2: Unary operators

The binary and unary operators supported are listed in table 4.1 and 4.2 respectively. Associated with them is their signature, indicating on which values they are defined. The operators are generalized to values by using the coercion function from the previous section. E.g. given values x and y

$$x \oplus y = c_{\beta, \text{Value}}(c_{\text{Value}, \alpha}(x) \oplus c_{\text{Value}, \alpha}(y))$$

where $\oplus : \alpha \times \alpha \rightarrow \beta$. Likewise can the unary operation be generalized to values

$$\circ x = c_{\alpha, \text{Value}}(\circ(c_{\text{Value}, \alpha}(x)))$$

where $\circ : \alpha \rightarrow \alpha$. E.g., performing addition `41 + true` first coerces the right side to `1` yielding `42` as result of the addition.

In table 4.1 logical **AND** and **OR** operators have two different notations. This is due to PHP specifying different precedence for the two notations. The textual operators bind weaker than assignments whereas the symbol operators bind stronger than assignments. Since the AST for the analysis is provided this difference have no direct impact on the analysis implementation.

For numeric operators the variables x , y , a , and b are used. x and y denote an integer larger than zero, $x, y \in \mathbb{Z} \wedge x, y > 0$ and a , b is a double smaller than zero or not a integer, $a, b \in \mathbb{R} \wedge (a, b < 0 \vee a, b \notin \mathbb{Z})$. Or an `uInt` number and `notUInt` number respectively, e.g. with this notion the addition $x + a$ would be an addition of a `uInt` number (x) and a `notUInt` number (a).

Table 4.3 defines abstract subtraction. Numbers in `uInt` are split into 0 and all other numbers while numbers in `notUInt` are split into negative and positive numbers. These splits are made to heighten the precision of the operator. 0 is the right-identity of subtraction which is used in the 0-column of table 4.3 to get `uInt` instead of \top . Subtracting negative numbers correspond to adding the absolute value of the number. For `uInt` this means that adding negative integers results in `uInt` instead of \top .

$-$	\perp	0	y	<code>uInt</code>	$b \in \mathbb{Z}$	b	<code>notUInt</code>	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	0	y	\top	$-b$	$-b$	\top	\top
x	\perp	x	$x - y$	\top	$x - b$	$x - b$	\top	\top
<code>uInt</code>	\perp	<code>uInt</code>	\top	\top	<code>uInt</code>	<code>notUInt</code>	\top	\top
$a \in \mathbb{Z}$	\perp	a	$a - y$	<code>notUInt</code>	$a - b$	$a - b$	\top	\top
a	\perp	a	$a - y$	<code>notUInt</code>	$a - b$	$a - b$	\top	\top
<code>notUInt</code>	\perp	<code>notUInt</code>	<code>notUInt</code>	<code>notUInt</code>	\top	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top	\top	\top

Table 4.3: Abstract Subtraction

►Add examples on subtraction◄

For division and modulo operators an error can occur trying to divide by 0. PHP handles division by 0 by returning the boolean **false** value instead of a number. For this reason division and modulo operators are functions from numbers to a value as opposed to the rest of the numeric operators. The

shorthand f is used for the boolean **false** value to keep the table smaller. Only the `uInt` part of the number lattice can contain 0 which restricts the amount of possible **false** returns for division. However as seen in table 4.4 the amount of possible **false** values are high for the modulus operator. This is because PHP handles modulus for decimal numbers by removing the decimal part which means $-0.5 \Rightarrow 0$ and $0.8 \Rightarrow 0$. The result of the modulus operator is always an integer and the sign is dictated by the sign of the left-hand operand which can be seen in the table by the \top -element column not consisting solely of \top -element results.

%	\perp	0	y	<code>uInt</code>	$1 > b > -1$	b	<code>notUInt</code>	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	f	0	$0 \sqcup f$	$0 \sqcup f$	0	$0 \sqcup f$	$0 \sqcup f$
x	\perp	f	$x \% y$	$\text{uInt} \sqcup f$	$x \% b$	$x \% b$	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$
<code>uInt</code>	\perp	f	<code>uInt</code>	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$	<code>uInt</code>	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$
$1 > a > -1$	\perp	f	$a \% y$	$\text{uInt} \sqcup f$	$a \% b$	$a \% b$	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$
a	\perp	f	$a \% y$	$\text{notUInt} \sqcup f$	$a \% b$	$a \% b$	$\text{notUInt} \sqcup f$	$\text{notUInt} \sqcup f$
<code>notUInt</code>	\perp	f	\top	$\top \sqcup f$	$\top \sqcup f$	\top	$\top \sqcup f$	$\top \sqcup f$
\top	\perp	f	\top	$\top \sqcup f$	$\top \sqcup f$	\top	$\top \sqcup f$	$\top \sqcup f$

Table 4.4: Abstract Modulus

Comparison operators are defined directly on values since different types can be compared with each other courtesy of type translation. PHP has an non-complete ordered definition of how values are compared depending on their type which can be seen in table 4.5. Any combination of operators not present in the table considered unspecified and yields a boolean \top value.

For performance reasons the analysis do not try to compare all different combinations of possible types when performing abstract comparison operators. If either value have multiple possible types the result is the boolean \top -element. Otherwise the comparison operators follow the order of table 4.5 for coercion of values and comparison of specific types are specified in the tables in appendix B.

Arrays are first of all compared by size. Equal sized arrays with different keys are incomparable and otherwise the arrays are compared by their values. Since the array lattice have no notion of size it is not possible to reason about the results of array comparisons. The only possible sound result is the boolean \top -element.

Type of left operand	Type of right operand	Result
null or string	string	null is coerced to string
bool or null	anything	operands are coerced to bool
string or number	string or number	strings are coerced to numbers
array	anything	arrays are always greater

Table 4.5: Comparison with Various Types based on [?]

4.7 The Monotone Framework

In order to perform a data flow analysis on a P0 program, the embellished monotone framework, as introduced by Nielson, Nielson and Hanikin, is used ►ref◄. A monotone data-flow analysis is a tuple, (L, \mathcal{F}) , where L is a lattice and \mathcal{F} is a monotone function space on L .

Given a program, an corresponding instance of the data-flow analysis can be expressed as a six-tuple, $(L, \mathcal{F}, F, E', \iota, f)$. Here L and \mathcal{F} are the lattice and function space of the analysis, F is a set of tuples expressing the between labels, E' is the set of external labels, ι is the initial lattice element of the external labels, and f is a mapping from labels to functions in \mathcal{F} . Labels are in this analysis defined as the product of nodes and context, $\mathcal{L} = \mathcal{N} \times \Delta$. From an instance, the data-flow equations, $A = \mathcal{L} \rightarrow L$, are defined as

$$A_{\bullet}((n, \delta)) = \begin{cases} f_{(n, \delta, (\delta, c))}(A_{\circ}((c, \delta)), A_{\circ}((n, \delta))) & \text{if } n = \text{return}_c(_) \\ f_{(n, \delta)}(A_{\circ}((n, \delta))) & \text{else} \end{cases} \quad (4.43)$$

where

$$A_{\circ}(l) = \bigsqcup \{A_{\bullet}(l') \mid (l', l) \in F\} \sqcup \iota_{E'}^l \quad (4.44)$$

and

$$\iota_{E'}^l = \begin{cases} \iota & \text{if } l \in E' \\ \perp & \text{else} \end{cases} \quad (4.45)$$

intuitively A_{\bullet} expresses the abstract state immediately after *performing* the given label and A_{\circ} the state just before. Solving these equations preforms the data-flow analysis.

Given a control flow graph, $G = (V, E, s, t)$, an instance of the analysis, where $L = \text{AnalysisLattice}$ and \mathcal{F} is the function space of the transfer functions, can be derived. Here f is defined throughout section 4.4, $F : \mathcal{L} \times \mathcal{L}$ is defined as

$$F = \{((n, \delta), (n', \delta')) \mid (n, n') \in E \wedge \delta \in \Delta \\ \wedge \text{validSuccessor}(n, n') \\ \wedge \delta' = \text{nextC}(n, \delta)\} \quad (4.46)$$

where

$$\text{nextC}(n, \delta) = \begin{cases} \delta' & \text{if } n = \text{exit}(_) \text{ and } \delta = (\delta', c) \\ (\delta, n) & \text{if } n = \text{call}(_) \\ \delta & \text{else} \end{cases} \quad (4.47)$$

The predicate *validSuccessor* is defined by definition 7, and in is necessary because of the ambiguity associated with potentially multiple outgoing edges of the *exit* node, which indicates that the flow may go from a exit node to an arbitrary result node.

This is naturally not the case. A successor, w , to a node, v , is only valid iff. v is an exit node and w is the return node corresponding to the call node of the current function-call, or if v is not an exit node.

Definition 7. The predicate $validSuccessor : \mathcal{L} \times \mathcal{L}$, is defined as

$$validSuccessor(n, \delta, n', \delta') \Leftrightarrow (n = exit(_) \wedge n' = result_c(_) \wedge \delta = (\delta'c)) \vee n \neq exit(_) \quad (4.48)$$

The set of external nodes is $E' = \{s\}$, the initial lattice element, ι , is the element where the empty context, Λ , maps to the state, $s_\iota = (\perp, g_\iota, h_\iota, \perp, \perp)$. Here the global scope, g_ι models the super-globals as

$$\begin{aligned} g_\iota = [& \$GLOBALS \mapsto \{HLoc(\Lambda, s, 0)\}, \\ & __SERVER \mapsto \{HLoc(\Lambda, s, 1)\}, \\ & __SESSION \mapsto \{HLoc(\Lambda, s, 2)\}, \\ & __ENV \mapsto \{HLoc(\Lambda, s, 3)\}, \\ & __COOKIE \mapsto \{HLoc(\Lambda, s, 4)\}, \\ & __POST \mapsto \{HLoc(\Lambda, s, 5)\}, \\ & __GET \mapsto \{HLoc(\Lambda, s, 6)\}, \\ & __REQUEST \mapsto \{HLoc(\Lambda, s, 7)\}, \\ & __FILES \mapsto \{HLoc(\Lambda, s, 8)\}] \end{aligned}$$

and the heap, h_ι is

$$\begin{aligned} h_\iota = [& HLoc(\Lambda, s, i \in [0, 2]) \mapsto Value(Array(\top)), \\ & HLoc(\Lambda, s, 3) \mapsto Value(ArrayMap([Index(\top) \mapsto HLoc(\Lambda, s, 9)])), \\ & HLoc(\Lambda, s, 4) \mapsto Value(ArrayMap([Index(\top) \mapsto HLoc(\Lambda, s, 10)])), \\ & HLoc(\Lambda, s, 5) \mapsto Value(ArrayMap([Index(\top) \mapsto HLoc(\Lambda, s, 11)])), \\ & HLoc(\Lambda, s, 6) \mapsto Value(ArrayMap([Index(\top) \mapsto HLoc(\Lambda, s, 12)])), \\ & HLoc(\Lambda, s, 7) \mapsto Value(ArrayMap([Index(\top) \mapsto HLoc(\Lambda, s, 13)])), \\ & HLoc(\Lambda, s, 8) \mapsto Value(ArrayMap([Index(\top) \mapsto HLoc(\Lambda, s, 14)])), \\ & HLoc(\Lambda, s, i \in [9, 10]) \mapsto Value(String(\top)), \\ & HLoc(\Lambda, s, i \in [11, 13]) \mapsto Value(Array(\top), String(\top)), \\ & HLoc(\Lambda, s, 14) \mapsto Value(ArrayMap([Index(\top) \mapsto HLoc(\Lambda, s, 15)])), \\ & HLoc(\Lambda, s, 15) \mapsto Value(Number(\top), String(\top)), \\ & _ \mapsto Value(Null(\top))] \end{aligned}$$

the last entry maps all other locations to the null value.

►Elaborate on decision of super-globals◄

4.8 Implementation

In order to solve the data-flow equation of the monotone framework, the above lattice, control-flow-graph, and transfer functions has been implemented in approximately 8100 lines of Java code, as a plug-in in the IntelliJ IDEA (Ultimate edition) development environment, by JetBrains►ref◄. This IDE supports multiple languages, such as Java, Python, C, C++, C#, Ruby, and PHP, with

tools for re-factoring, type-checking, a vast library of plug-ins, developed by JetBrains or the JetBrains community, etc.

When running a plug-in on a given program, an AST and type-information is available from the environment, expressed as **PSIElements** (Program-Structure-Interface elements). The control-flow graph is created from a single pass parsing of these elements and each node is keeping a reference to the element, from which they were created. This allows for easy error reporting, when performing the analysis.

The lattices are defined by interfaces and the elements are implemented as immutable data structures of these interfaces. In the name of efficiency the domain of map-lattice elements (**MapLatticeElement**) are defined as the indices of modified entries, not including the values defaulted to $\text{Value}(\text{Null}(\top))$ in the initial array. Comparing two map elements is then done by comparing the values corresponding to the joint domain of the elements, which in turn allows comparisons to be done in finite time. This *short-cut* does also entail that when updating the entry of a \top -array, only the variables initialized are effected, which is sound and yields a more precise model.

The transfer functions are implemented notoriously as introduced in section 4.4, with added statements for reporting of suspicious behaviour, to the IDE through **Annotation**'s. Reporting does not effect the outcome of the analysis and is made possible by the references to the **PSIElements** in the nodes of the control-flow-graph. The analysis reports the following

- Appending on definite map yields error
- Merging a list with a map and visa versa yields error
- Definite string write on list yields error
- Adding new type to array yields warning
- Writing to possible non-array yields warning
- Appending on possible map yields warning
- Possible string write on list yields warning

►**Write better list**◄ Solving the data-flow equations are performed by an implementation of the worklist algorithm (Algorithm 1). Notice, that since $\iota \not\sqsubseteq \perp$, any reachable node is bound to be analysed at least once.

4.8.1 Library functions

Until now it has been assumed that any function called should be implemented in the same program. PHP however comes with a large number of internal functions, which are essential any program. In order to support these, a **LibraryFunction** interface has been created. Implementing a class of this interface, and registering it in the **PSIParser**, allows modelling of a function by expressing the signature, whether arguments and return values are passed by reference or value, and specifying a transfer function for the result node.

Algorithm 1 Worklist algorithm

Require: Control flow graph, $G = (V, E, s, t)$

- 1: $I = [\forall \delta \in \Delta, n \in \mathcal{N}.(n, \delta) \mapsto \perp]$
- 2: $I[(s, \Lambda) \mapsto \iota]$
- 3: $W = [f \in F | f = ((s, \Lambda), _)]$
- 4: **while** $W \neq \emptyset$ **do**
- 5: $(l_1, l_2) = W.takeFirst()$
- 6: **if** $f_{l_1}(l_1) \not\sqsubseteq I[l_2]$ **then**
- 7: $I[l_2 \mapsto f_{l_1}(l_1)]$
- 8: $W.append([f \in F | f = (l_2, _)])$
- 9: **end if**
- 10: **end while**

Let lib be the name of a library function, then the transfer function of the $call_{lib}(_)$ node becomes the identity function, the flow graph is the graph containing a *start* node followed by an *exit* node, and the transfer function of the $result_{call_{lib}(_)}$ is the specified transfer function.

For the library function $fn = \text{array_merge}$ **►ref:** <http://php.net/manual/en/function.array-merge.php> **◄** the arguments and return value are all passed by value. The transfer function for the result node, $n = result_c(c_{tar})$, where $c = call_{fn}(t_1, \dots, t_n)$, is defined as

$$\begin{aligned}
 f_{n, \delta_{call}, \delta_{exit}}(l_{call}, l_{exit}) = & \text{let } (s_l, s_g, s_h, s_t, s_{ht}) = l_{call}[\delta_{call}] \text{ in} \\
 & \text{let } V = \{v_i | v_i = s_t[t_i] \wedge i \in [1; n]\} \text{ in} \\
 & \text{let } (v_a, v_s, v_n, v_b, v_u) = \bigsqcup V \text{ in} \\
 & \text{let } w = \\
 & \quad \text{if } \perp \in V \vee v_s \neq \perp \vee v_n \neq \perp \\
 & \quad \vee v_b \neq \perp \vee v_u \neq \perp \text{ then} \\
 & \quad \text{Value(Null}(\top)) \text{ else } \perp \\
 & \text{in} \\
 & \text{let } v = w \sqcup \text{Value}(v_a) \text{ in} \\
 & \text{let } (s'_t, s'_{ht}, s'_h) = \text{match } c_{tar} \\
 & \quad \text{with Temp: } (s_t[c_{tar} \mapsto v], s_{ht}, s_h) \\
 & \quad \text{with HeapTemp:} \\
 & \quad (s_t, \\
 & \quad \quad s_{ht}[c_{tar} \mapsto \text{HLoc}(\delta_{call}, n, 0)], \\
 & \quad \quad s_h[\text{HLoc}(\delta_{call}, n, 0) \mapsto v]) \\
 & \text{in} \\
 & l_{call}[\delta_{call} \mapsto (s_l, s_g, s'_h, s'_t, s'_{ht})]
 \end{aligned}$$

This basically just joins the arrays of each input value, taking invalid arguments yielding **null**, into account. The joined value is then stored in the heap or temporary storage, depending on the result argument.

The implementation has been tested against 48 small functional tests. While these are not representative of actual PHP programs, neither in size or feature usage, they do aim to cover as many feature cases as possible.

►Elaborate on more library functions.◄

Chapter 5

Case Study

To evaluate the effect of the analysis developed this chapter studies an amount of cases and how the analysis applies to those cases. Each case consists of a small PHP program inspired by code from a real PHP application accompanied by the result of running the analysis on the case program, how the analysis reached the results and how the results can be used.

►Add some text on how the analysis ran. E.g. speed, memory, etc.◄

5.1 Maps and lists

This study concerns a constructed case to illustrate how the analysis work with the two array-types, lists and maps. What the program does is define an array of month names and use a loop to create an array with these month names as keys and the corresponding month number as value as seen in listing 5.1. The first array is considered a list by the analysis whereas the second array is considered a map from strings to numbers. The last two lines of the program uses the map to look up the number of a month name provided by the user (from the superglobal `$_GET`). Since the definition of lists in the analysis does not take indices into account the analysis lose precision for the map in this example. The analysis can only tell it is a map from strings to unsigned integers and nothing more precise.

The first step of the analysis is generating the control flow graph. For the array in the first line of this program a CFG node is generated for creating an empty array. Then nodes are generated for each of the 12 initial strings as well as for adding them to the array, when the analysis encounter on of these nodes the empty array will become a list type array since they are represented as array append nodes due to no keys being specified. Lastly the variable name “monthNames” will be associated with the initialized list array.

The next interesting step of the analysis is line 5 where an array write node is generated preceded by a sub tree for the expression `array_pop($monthNames)`. When the analysis encounters the array write node it determines the possible index values for the expression sub tree and since string values are possible the array becomes a map array.

Program 5.1 Turning a month list into a “month name to month number”-map

```
1 $monthNames = [ "January", "February", "March", "April",  
    , "May", "June", "July", "August", "September", "  
    October", "November", "December" ];  
2 $monthMap = [];  
3  
4 for ($i = 1; $i <= 12; $i++) {  
5     $monthMap[array_pop($monthNames)] = $i;  
6 }  
7  
8 $input = $_GET["monthName"];  
9  
10 echo $input . " is month number " . $monthMap[$input];
```

►write something about GET◄

5.2 Array Pivot

The following example is found in an array helper class in the Joomla content management system. The code is adapted to fit in P0 as well as shortened to only display necessary code. The function accepts an array as input and return an array with the unique values of the input array as keys and the keys of the input array as values. The odd thing about this function is how unique and duplicate values from the input array are represented differently in the output array namely as either a string/integer value or an array of string/integer values. See listing 5.3 for an example. This mix of arrays and scalar values in the resulting array forces the user of the function to check the data type before using the values. Necessary data type checks strongly suggests a bad design and the analysis provides warnings about suspicious use of arrays as seen in figure 5.1. Both warnings are based on the fact that values of the array associated with “result” might be arrays or scalar values.

►the warning is due to path insensitivity, and not directly caused by the actual problem, we have to describe that somehow and argue that capturing the problem indirectly is good enough◄

```
else if ($counter[$resultKey] == 1)
{
    // If there is a second time, we convert the value into an array.
    $result[$resultKey] = [$result[$resultKey], $resultValue];
    $counter[$resultKey]++;
}
else
{
    // After the second time, no need to track any more. Just
    $result[$resultKey][] = $resultValue;
}
```

Figure 5.1: Warnings shown due to the mixed value types of the result array

Program 5.2 Pivoting an array so values becomes keys and keys becomes values

```

1 function pivot($source)
2 {
3     $result = [];
4     $counter = [];
5
6     for ($i = 0; $i < count($source); $i++)
7     {
8         $rKey = $source[$i];
9         $rValue = $i;
10
11         if (empty($counter[$rKey]))
12         {
13             $result[$rKey] = $rValue;
14             $counter[$rKey] = 1;
15         }
16         else if ($counter[$rKey] == 1)
17         {
18             $result[$rKey] = [$result[$rKey],
19                             $rValue];
20             $counter[$rKey]++;
21         }
22         else
23         {
24             $result[$rKey][] = $rValue;
25         }
26     }
27     return $result;
28 }
29
30 $simpleArr =
31     [1,2,3,4,5,6,7,8,1,5,3,7,9,0,4,2,5,8,4,3,8,9];
32 $result = pivot($simpleArr);

```

Program 5.3 Example of input and output of the pivot function

```

1 [1,2,3,4,5,4,3,2,1]

1 [1 => [0,8],
2  2 => [1,7],
3  3 => [2,6],
4  4 => [3,5],
5  5 => 4]

```

►array write warning does not consider possible null values since that will cause false positives◄

5.3 Directory Content

Inspired by the framework CodeIgniter this example demonstrates mixing of the map and list type arrays. The function takes a directory name and a depth as input and returns the structure of the given directory recursively until the given depth is reached or no more sub directories are found. The resulting structure is a mix of a list of file names and a map from sub directory name to the structure of that sub directory. Mixing a list array and a map array like seen in listing 5.4 provides no obvious way of using the output of the function. If the function ever has to be refactored the users may use the resulting structure in many different ways making it difficult to refactor without breaking possible use cases.

The analysis provides an error to bring attention to the possibly bad design choice as seen in figure 5.2. The mixing of array types are detected by the analysis because the append operation creates a list array and then afterwards a write is encountered with the index possibly being a string which then triggers the error.

Another possible problem with this design is that directories can be named with just numbers which might then override a file already added to the list. This problem occurs because the library function `readdir` follows the order of which the file system stores the files and directories. Such an error is difficult to reproduce and therefore potentially difficult to find.

```
if (($directory_depth < 1 || $new_depth > 0) && is_dir($source_dir.$file))
{
    $filedata[$file] = directory_map($source_dir.$file, $new_depth);
}
else
{
    $filedata[] = $file;
}
```

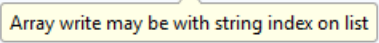


Figure 5.2: Error about writing to a list, thus mixing the list and map type

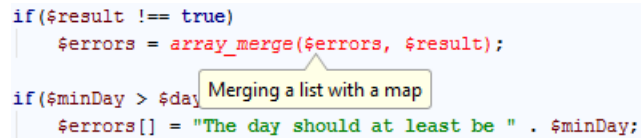
5.4 Date Validation

Similarly to the previous case this case also concern mixing of map and list arrays. The case is inspired by the e-commerce platform Magento where code similar to the two functions seen in 5.5 can be found. The problem can easily be spotted when placing the two functions together as done here, however the original code is separated in two different files which conceals the problem.

The difference from the previous example is that the mixing occurs in a library function `array_merge`. Each library function has transfer function which implements an abstract version of that particular function. The `array_merge`

Program 5.4 Mixing of map and list

```
1 function directory_map($dir , $depth)
2 {
3     if ($fp = opendir($dir))
4     {
5         $filedata = [];
6         $new_depth = $depth - 1;
7         while (FALSE !== ($file = readdir($fp)))
8         {
9             is_dir($dir.$file) && $file .= '/';
10            if (($depth < 1 || $new_depth > 0)
11                && is_dir($dir.$file))
12            {
13                $filedata[$file] = directory_map($dir.
14                $file , $new_depth);
15            }
16            else
17            {
18                $filedata[] = $file;
19            }
20        }
21        closedir($fp);
22        return $filedata;
23    }
24    return FALSE;
25 }
26 $result = directory_map("testDir" , 2);
```



The screenshot shows a snippet of PHP code. The first line is `if($result !== true)`. The second line is `$errors = array_merge($errors, $result);`. The third line is `if($minDay > $day`. The fourth line is `$errors[] = "The day should at least be " . $minDay;`. A yellow callout box with a pointer to the `$errors` variable in the second line contains the text "Merging a list with a map".

Figure 5.3: Array merge error message

abstract function also detects possible merges of different types of arrays and warns the user like shown in figure 5.3.

5.5 Keeping track of instances

This case demonstrates the opposite of the case in section 5.3. In lines 8 and 9 the arrays associated with “keyArray” and “valueArray” are used in a map context by writing to a string index. Then in lines 13 and 14 the array append operation is used with both arrays. When detecting the latter the analysis presents the user with an error as seen in figure 5.4 turning the attention of the user to the fact that she is about to append to an array previously used in a map context. The program incidentally functions correctly even though half the program intends to use the array associated with “keyArray” as a map and the other half as a list. Maintaining this code is difficult since the intention of the code is not clear and putting attention to that enables the mistake to be

Program 5.5 Date Validation split into different functions resulting in a mixed map/list output

```
1 function isValidDate($day, $month, $year) {
2     $errors = [];
3     if(!checkdate($month, $day, $year))
4         $errors["invalidDate"] = "The given date is
        not valid";
5
6     return $errors;
7 }
8
9 function isMinimumDay($day, $month, $year, $required,
    $minDay) {
10     $errors = [];
11     if($required && $day == 0 && $month == 0 && $year
        == 0)
12         $errors[] = "The date is required";
13
14     $result = isValidDate($day, $month, $year);
15     $errors = array_merge($errors, $result);
16
17     if($minDay > $day)
18         $errors[] = "The day should at least be " .
        $minDay;
19
20     return $errors;
21 }
22 $valid = isMinimumDay(27, 5, 2015, true, 6);
23 $invalid = isMinimumDay(1, 3, 1991, true, 5);
```

```

$keyArray[] = $instance;
return $valueArray[] = $value;

```

Appending on map

Figure 5.4: Error message provided when using array append with a map array

fixed right away.

Program 5.6

```

1 $keyArray = [];
2 $valueArray = [];
3 function createInstance($string, $instance, $value)
4 {
5     global $keyArray, $valueArray;
6
7     if (!isset($keyArray[$string])) {
8         $keyArray[$string] = [];
9         $valueArray[$string] = [];
10    } else if (($k = array_search($instance, $keyArray,
11        true)) !== false) {
12        return $this->valueArray[$k];
13    }
14    $keyArray[] = $instance;
15    return $valueArray[] = $value;
16 }
17 createInstance("test", "test2", "testValue");

```

5.6 Joining array values

This case is inspired by [►what?◄](#) and shows how the analysis yields an error when using the array function `array_pop` on a map type array. The program seen in listing 5.7 builds a map from a person to an animal while also adding in a separator. The separators are then used when the map values are turned into a string using the `implode` function. The map can afterwards be used to refer to which animal belong to which person. To remove the unwanted separator at the end of the map `array_pop` is used to remove the last element of the array. The analysis then alerts the user to the fact that `array_pop` is used with a map which is suspicious. The separator should instead be specified as the first argument for `implode` which avoids the use of `array_pop` altogether.

```

array_pop($animalMap);
$animalString .= implode(" ", $animalMap);
echo "The following animals: " . $animalString;

```

array_pop on map

Figure 5.5: Using the pop function on a map array

5.7 Conclusion

Program 5.7 Joining array values to a string

```
1 $people = [ "John", "Jane", "Alice", "Bob" ];
2 $animals = [ "Dog", "Cat", "Bird", "Fish" ];
3 $animalMap = [ ];
4
5 for ($i = 0; $i < count($people); $i++) {
6     $animalMap[$people[$i]] = $animals[$i];
7     $animalMap[$i] = ", ";
8 }
9
10 array_pop($animalMap);
11 $animalString = implode("", $animalMap);
12 echo "The people have the following animals: " .
    $animalString;
```

►Add small conclusion on cases◄

Chapter 6

Related work

6.1 Dynamic features

In this thesis a lot of the dynamic features of PHP have not been covered even though [?] and [?] confirms that dynamic features are used in most PHP programs.

Programs composed of multiple files using **require** and **include** are considered by [?] which provides a technique for statically resolving includes in PHP. Using first a file-centered contextless algorithm to find possible file matches and afterwards refining the results with a context sensitive program-centered algorithm.

WeVerca the framework developed by [?] abstract away the dynamic dispatch and variable variables uses by providing the control-flow as well as the heap shape and dynamic data access directly to the user of the framework. WeVerca uses the Phalanger parser which currently supports PHP features up to PHP 5.4 with support for newer features in development. With the aim of PHP 5.6 support for the analysis in this thesis WeVerca was discarded as an option early on.

6.2 Static Analysis

JavaScript is similar to PHP in many ways. The static analysis provided by this thesis is inspired by TAJIS [?], a type analysis for JavaScript. **►talk about differences between TAJIS and TAPAS◄**

Pixy [?] is a static analysis tool aimed at detecting security vulnerabilities. The tool is limited to PHP 4 which is a long out-dated version of PHP neither developed nor supported anymore. The analysis developed in this thesis aims to support PHP 5.6 which is the newest version as of the time of writing.

6.3 PHP References

The concept of references in PHP complicates reasoning about PHP programs. In [?] the copy-on-write practice exhibited by the official PHP interpreter in

investigated and compared to the stated copy-on-assignment semantics of the PHP documentation.

References in PHP arrays are also treated by [?] implementing a data-flow analysis using the concept of aliases to handle references.

6.4 Static Approximation of Dynamically Generated Web Pages

►write summary◄ [?]

6.5 Static Detection of Cross-Site Scripting Vulnerabilities

►write summary◄ [?]

6.6 Static Detection of Security Vulnerabilities in Scripting Languages

►write summary◄ [?]

6.7 Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking

►write summary◄ [?]

6.8 Sound and Precise Analysis of Web Applications for Injection Vulnerabilities

►write summary◄ [?]

6.9 Alias Analysis for Object-Oriented Programs

►write summary◄ [?]

6.10 Two Approaches to Interprocedural Data Flow Analysis

►write summary◄ [?]

6.11 Practical Blended Taint Analysis for JavaScript

Method: Use JSBAF to make a blended taint analysis

Important points: static analyses are slow (> 10 minutes), blended analysis is much faster since impossible or unused paths can be pruned. More problems can be identified. Fewer false alarms [?]

6.12 Blended Analysis for Performance Understanding of Framework-based Applications

Important points: blended analysis is good when you have a limited amount of possible inputs. [?]

Chapter 7

Conclusion



Chapter 8

Future Work

► Adding variable-variables with expressing scope as array ◄ ► Handling read of \top -element arrays when returning location sets, current solution is naively returning the whole heap ◄
► ... ◄

Appendix A

Basic Definitions

Definition 8. A partial order, $S = (A, \sqsubseteq)$ is a set of elements, A , and a binary relation, $\sqsubseteq: A \times A$, where \sqsubseteq is

- Reflexive: $\forall a \in A : a \sqsubseteq a$
- Anti-symmetric: $\forall a, b \in A : a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$
- Transitive: $\forall a, b, c \in A : a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$

Definition 9. A lattice $L = (E, \sqsubseteq)$ is a partial order where each subset $S \subseteq E$ has a least upper bound and a greatest lower bound. E.g. $\sqcup S$ and $\sqcap S$ respectively

Definition 10. The sum of two lattices, $L_1 = (E_1, \sqsubseteq_1)$ and $L_2 = (E_2, \sqsubseteq_2)$ where $\{\top, \perp\} \subseteq E_1 \cap E_2$, is defined as

$$L_{Sum} = L_1 + L_2 = (E_{Sum}, \sqsubseteq_{Sum}) \quad (\text{A.1})$$

Where

$$E_{Sum} = \{(i, x) | x \in L_i \setminus \{\top, \perp\}\} \cup \{\top, \perp\} \quad (\text{A.2})$$

and for every $e_1, e_2 \in E_{Sum}$

$$e_1 \sqsubseteq_{Sum} e_2 \Leftrightarrow (x \sqsubseteq_i y \wedge e_1 = (x, i) \wedge e_2 = (y, i)) \vee e_2 = \top \vee e_1 = \perp \quad (\text{A.3})$$

►**Lemma:** L_{Sum} is a lattice. ◀

Definition 11. The product of two lattices, $L_1 = (E_1, \sqsubseteq_1)$ and $L_2 = (E_2, \sqsubseteq_2)$, are defined as

$$L_{Prod} = L_1 \times L_2 = (E_{Prod}, \sqsubseteq_{Prod}) \quad (\text{A.4})$$

Where

$$E_{Prod} = \{(e_1, e_2) | e_1 \in L_1, e_2 \in L_2\} \quad (\text{A.5})$$

and for every $(x_1, x_2), (y_1, y_2) \in E_{Prod}$

$$e_1 \sqsubseteq e_2 \Leftrightarrow x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2 \quad (\text{A.6})$$

►**Lemma:** L_{Prod} is a lattice. ◀

Definition 12. Given a set, $A = \{a_1, \dots, a_n\}$, and a lattice $L = \{E, \sqsubseteq\}$, a map lattice is defined as

$$L_{Map} = A \mapsto L = (E_{Map}, \sqsubseteq_{Map}) \quad (\text{A.7})$$

Where

$$E_{Map} = \{([a_1 \mapsto x_1, \dots, a_n \mapsto x_n] | x_i \in E)\} \quad (\text{A.8})$$

and for two elements $e_1, e_2 \in E_{Map}$,

$$e_1 \sqsubseteq_{Map} e_2 \Leftrightarrow \forall a \in A : e_1(a) \sqsubseteq e_2(a) \quad (\text{A.9})$$

►**Introduce reading set of locations from heap**◀

►**Lemma:** L_{Map} is a lattice. ◀

Definition 13. Given a set A , the powerset is defined as

$$P(A) = (E_P, \sqsubseteq_P) \quad (\text{A.10})$$

Where

$$E_P = \{S | S \subseteq E\} \quad (\text{A.11})$$

and for two elements $e_1, e_2 \in E_P$

$$e_1 \sqsubseteq_P e_2 \Leftrightarrow e_1 \subseteq e_2 \quad (\text{A.12})$$

►**Lemma:** $P(A)$ is a lattice. ◀

Appendix B

Abstract Operators

The following definitions are used for all tables in this appendix.

$$x, y \in \mathbb{Z} \wedge x, y > 0$$

$$a, b \in \mathbb{R} \wedge (a, b < 0 \vee a, b \notin \mathbb{Z})$$

Furthermore the shorthand f is used for the boolean **false** value.

$+$	\perp	0	y	uInt	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	0	y	uInt	b	notUInt	\top
x	\perp	x	$x + y$	uInt	$x + b$	\top	\top
uInt	\perp	uInt	uInt	uInt	\top	\top	\top
a	\perp	a	$a + y$	\top	$a + b$	\top	\top
notUInt	\perp	notUInt	\top	\top	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top	\top

Table B.1: Abstract addition

$-$	\perp	0	y	uInt	$b < 0 \wedge b \in \mathbb{Z}$	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	0	y	\top	$-b$	$-b$	\top	\top
x	\perp	x	$x - y$	\top	$x - b$	$x - b$	\top	\top
uInt	\perp	uInt	\top	\top	uInt	notUInt	\top	\top
$a < 0 \wedge a \in \mathbb{Z}$	\perp	a	$a - y$	notUInt	$a - b$	$a - b$	\top	\top
a	\perp	a	$a - y$	notUInt	$a - b$	$a - b$	\top	\top
notUInt	\perp	notUInt	notUInt	notUInt	\top	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top	\top	\top

Table B.2: Abstract subtraction

\cdot	\perp	0	y	uInt	$b < 0$	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	0	0	0	0	0	0	0
x	\perp	0	$x \cdot y$	uInt	$x \cdot b$	$x \cdot b$	\top	\top
uInt	\perp	0	uInt	uInt	notUInt	\top	\top	\top
$a < 0$	\perp	0	$a \cdot y$	notUInt	\top	$a \cdot b$	\top	\top
a	\perp	0	$a \cdot y$	\top	$a \cdot b$	$a \cdot b$	\top	\top
notUInt	\perp	0	\top	\top	\top	\top	\top	\top
\top	\perp	0	\top	\top	\top	\top	\top	\top

Table B.3: Abstract multiplication

$/$	\perp	0	y	uInt	$b < 0$	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	f	0	$0 \sqcup f$	0	0	0	$0 \sqcup f$
x	\perp	f	$\frac{x}{y}$	$\top \sqcup f$	$\frac{x}{b}$	$\frac{x}{b}$	\top	$\top \sqcup f$
uInt	\perp	f	\top	$\top \sqcup f$	notUInt	\top	\top	$\top \sqcup f$
$a < 0$	\perp	f	$\frac{a}{y}$	notUInt $\sqcup f$	$\frac{a}{b}$	$\frac{a}{b}$	\top	$\top \sqcup f$
a	\perp	f	$\frac{a}{y}$	$\top \sqcup f$	$\frac{a}{b}$	$\frac{a}{b}$	\top	$\top \sqcup f$
notUInt	\perp	f	\top	$\top \sqcup f$	\top	\top	\top	$\top \sqcup f$
\top	\perp	f	$\top \sqcup f$	$\top \sqcup f$	\top	\top	\top	$\top \sqcup f$

Table B.4: Abstract division

$\%$	\perp	0	y	uInt	$b > -1$	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	f	0	$0 \sqcup f$	$0 \sqcup f$	0	$0 \sqcup f$	$0 \sqcup f$
x	\perp	f	$x \% y$	uInt $\sqcup f$	$x \% b$	$x \% b$	uInt $\sqcup f$	uInt $\sqcup f$
uInt	\perp	f	uInt	uInt $\sqcup f$	uInt $\sqcup f$	uInt	uInt $\sqcup f$	uInt $\sqcup f$
$a > -1$	\perp	f	$a \% y$	uInt $\sqcup f$	$a \% b$	$a \% b$	uInt $\sqcup f$	uInt $\sqcup f$
a	\perp	f	$a \% y$	notUInt $\sqcup f$	$a \% b$	$a \% b$	notUInt $\sqcup f$	notUInt $\sqcup f$
notUInt	\perp	f	\top	$\top \sqcup f$	$\top \sqcup f$	\top	$\top \sqcup f$	$\top \sqcup f$
\top	\perp	f	\top	$\top \sqcup f$	$\top \sqcup f$	\top	$\top \sqcup f$	$\top \sqcup f$

Table B.5: Abstract modulus

$**$	\perp	0	y	uInt	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
0	\perp	1	0	uInt	0	0	uInt
x	\perp	1	x^y	uInt	\top	\top	\top
uInt	\perp	1	uInt	uInt	\top	\top	\top
a	\perp	1	a^y	\top	a^b	\top	\top
notUInt	\perp	1	\top	\top	\top	\top	\top
\top	\perp	1	\top	\top	\top	\top	\top

Table B.6: Abstract exponentiation

$==$	\perp	t	f	\top	\neq	\perp	t	f	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
t	\perp	t	f	\top	t	\perp	f	t	\top
f	\perp	f	t	\top	f	\perp	t	f	\top
\top	\perp	\top	\top	\top	\top	\perp	\top	\top	\top

(a) Equality					(b) Not Equality				
$<$	\perp	t	f	\top	\leq	\perp	t	f	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
t	\perp	f	f	f	t	\perp	t	f	\top
f	\perp	t	f	\top	f	\perp	t	t	t
\top	\perp	\top	f	\top	\top	\perp	t	\top	\top

(c) Less Than					(d) Less Than/Equal				
$>$	\perp	t	f	\top	\leq	\perp	t	f	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
t	\perp	f	t	\top	t	\perp	t	t	t
f	\perp	f	f	f	f	\perp	f	t	\top
\top	\perp	f	\top	\top	\top	\perp	\top	t	\top

(e) Greater Than					(f) Greater Than/Equal				
$>$	\perp	t	f	\top	\geq	\perp	t	f	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
t	\perp	f	t	\top	t	\perp	t	t	t
f	\perp	f	f	f	f	\perp	f	t	\top
\top	\perp	f	\top	\top	\top	\perp	\top	t	\top

Table B.7: Abstract comparison operators on booleans

$==$	\perp	y	$uIntString$	s	$notUIntString$	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp
x	\perp	$x==y$	\top	f	f	\top
$uIntString$	\perp	\top	\top	f	f	\top
r	\perp	f	f	$r==s$	\top	\top
$notUIntString$	\perp	f	f	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top

Table B.8: Abstract Equal for Strings

\neq	\perp	y	$uIntString$	s	$notUIntString$	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp
x	\perp	$x!=y$	\top	t	t	\top
$uIntString$	\perp	\top	\top	t	t	\top
r	\perp	t	t	$r!=s$	\top	\top
$notUIntString$	\perp	t	t	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top

Table B.9: Abstract Not Equal for Strings

$\oplus \in \{<, >, \leq, \geq\}$	\perp	y	uIntString	s	notUIntString	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp
x	\perp	$x \oplus y$	\top	$x \oplus s$	\top	\top
uIntString	\perp	\top	\top	\top	\top	\top
r	\perp	$r \oplus y$	\top	$r \oplus s$	\top	\top
notUIntString	\perp	\top	\top	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top

Table B.10: Abstract less than (or equal) and greater than (or equal) for on strings.

$==$	\perp	y	uInt	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp
x	\perp	$x == y$	\top	f	f	\top
uInt	\perp	\top	\top	f	f	\top
a	\perp	f	f	$a == b$	\top	\top
notUInt	\perp	f	f	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top

Table B.11: Abstract equal on numbers

$!=$	\perp	y	uInt	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp
x	\perp	$x != y$	\top	t	t	\top
uInt	\perp	\top	\top	t	t	\top
a	\perp	t	t	$a != b$	\top	\top
notUInt	\perp	t	t	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top

Table B.12: Abstract not equal on numbers

$\oplus \in \{<, \leq\}$	\perp	y	uInt	b < 0	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
x	\perp	$x \oplus y$	\top	f	$x \oplus b$	\top	\top
uInt	\perp	\top	\top	f	\top	\top	\top
a < 0	\perp	t	t	$a \oplus b$	$a \oplus b$	\top	\top
a	\perp	$a \oplus y$	\top	$a \oplus b$	$a \oplus b$	\top	\top
notUInt	\perp	\top	\top	\top	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top	\top

Table B.13: Abstract less than (or equal) on numbers

$\oplus \in \{>, \geq\}$	\perp	y	uInt	$b < 0$	b	notUInt	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
x	\perp	$x \oplus y$	\top	t	$x \oplus b$	\top	\top
uInt	\perp	\top	\top	t	\top	\top	\top
$a < 0$	\perp	f	f	$a \oplus b$	$a \oplus b$	\top	\top
a	\perp	$a \oplus y$	\top	$a \oplus b$	$a \oplus b$	\top	\top
notUInt	\perp	\top	\top	\top	\top	\top	\top
\top	\perp	\top	\top	\top	\top	\top	\top

Table B.14: Abstract greater than (or equal) on numbers

&&	\perp	t	f	\top	 	\perp	t	f	\top
\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp
t	\perp	t	f	\top	t	\perp	t	t	t
f	\perp	f	f	f	f	\perp	t	f	\top
\top	\perp	\top	f	\top	\top	\perp	t	\top	\top

(a) **AND**

XOR	\perp	t	f	\top
\perp	\perp	\perp	\perp	\perp
t	\perp	f	t	\top
f	\perp	t	f	\top
\top	\perp	\top	\top	\top

(c) **XOR**

(b) **OR**

Table B.15: Abstract logical binary boolean operators

$++$	\perp	n	uInt	notUInt	\top
	\perp	$n + 1$	uInt	\top	\top

(a) Abstract pre- and post-increment

$--$	\perp	n	uInt	notUInt	\top
	\perp	$n - 1$	\top	notUInt	\top

(b) Abstract pre- and post-decrement

$-$	\perp	n	uInt	notUInt	\top
	\perp	$-n$	notUInt	\top	\top

(c) Unary minus

$!$	\perp	true	false	\top
	\perp	false	true	\top

(d) Boolean negate

Table B.16: Abstract logical binary boolean operators