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 9, 2015.

Contents

Chapter 1

Introduction

PHP is one of the most popular languages for server-side web-development. It is powering over 80% of the web[1], including major websites, such as Wikipedia and Facebook, and the most used CMS's: Wordpress, Joomla, Dupal, 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 using a static analysis. An *interprocedural data-flow type-analysis* is proposed to detect suspicious cross-use of the identified patterns, and consequently evaluated on live code.

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 possibles 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.

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 observers 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 $a[0] = 7; // Result: $a = [7,2,3]; $b = [7,2,3]; $c =
      7;
7 $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 $count(a) - 1$ 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 $count(a) - 1$ does not exists. 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 `$_SERVER` array[1]. This array contains server and execution environment information, which are of different types, e.g. `$_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 PHPUnit[2], a unit testing framework for PHP programs similar to JUnit for Java programs. By running the test suites on a modified PHP interpreter[3], 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 function[4], 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) $y` is an example of an assignment from a temporary variable. Here the the `$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       0       0x539
3 assign_tmp       1       a.php       NULL    array   1       1
        3       0x539
4 array_append     2       a.php       1       1       3       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, strings are logged only as `string`, booleans as 0 if `false` otherwise 1, and the null value logged as `NULL`. Strings are 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 useful 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 an execuation of a test suite, 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 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 uses on different 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 positionally equal, $x_1 \overset{pos}{=} x_2$, 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

$$\texttt{hash\_init}\ a_1$$

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

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

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 positionally 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, Vagrant[5] is used to create a clean and reproducible environment. A Vagrant initialization file[6] 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 of arrays relies on the type of the values read from or written to the arrays. This implies that there is no reasoning about arrays never read or written. Some of these arrays may fall into the uncategorized category but remain undetected by the analysis.
- Callables can be written as anonymous functions, strings, or arrays containing either an instance or a string representing a class name together with string represent a function name. Arrays of callables should be considered lists however this analysis will classify them as objects. Example 3.3 shows different types of callables.

[5] `http://vagrantup.com/`
[6] `http://github.com/Silwing/tapas-survey`

Program 3.3 Callables in PHP

```php
class A{

    public function f1(){
        ...
    }

}

function f(){
    ...
}

$a = new A();

$callable1 = [$a, "f1"];
$callable2 = ["A", "f1"];
$callable3 = "f";
$callable4 = function() use ($a){
    ...
};
```

- 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. Combining multiple arrays may lead to uncategorizable arrays even if each array is categorizable. A possible solution would be if the interpreter kept character location information in addition to line number and file name.

```
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`

```php
<?php

print_r([1,2,3]);

$a[1][2] = 3;
```

(3.3.2) File: `b.php`

Figure 3.3: A problematic program

3.3 Results

►Add something about file sizes of log-files◄ 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**►Not true. How is this relevant◄**. 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 **►Not true, it might coincidentally. Rewrite◄**. Since so few cyclic arrays were found a manual inspection of the code locations of the findings was possible. The manual inspection showed that almost all of the reported cyclic arrays are uses of the superglobal `$GLOBALS` which is an array consisting of all variables defined in the global scope. This array also has a reference to itself which makes it a cyclic array. Only two of the frameworks in the corpus had cases where it was not immediately clear whether it was a use of the `$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 `$GLOBALS` array passed to the function. The `$GLOBALS` array is a special case to handle **►How is it a special case?◄** even without taking its cyclic structure into account, so uses of this array can be disregarded when considering cyclic arrays **►How can it be disregarded?◄**. Based on the results of the manual inspection the hypothesis of arrays being acyclic is true for almost all frameworks in the corpus. **►Write about how no creation of cyclic arrays has been observed◄**.

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 as discussed in section 3.2.5. If multiple categorizable arrays from different categories are identified as

a single array the combined array ends up in the uncategorizable part of the distribution.

The Object group marked with List is by definition 3 categorized as objects, but they might fit better into the List category, as lists of a top level type **►What is 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. **►Introduce notion predictors, elaborate section◄**

	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

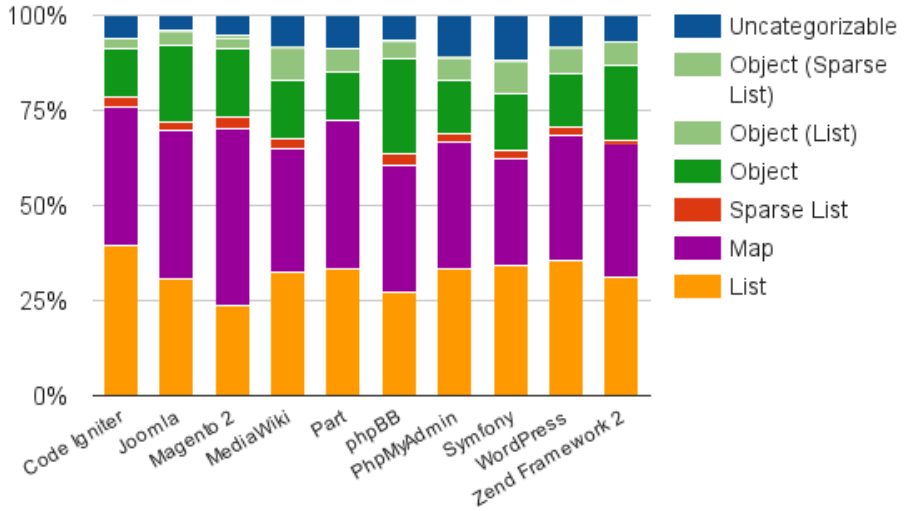


Figure 3.4: Distribution of different types of arrays

Figure 3.5 shows the distribution of operations on the arrays over the differ-

ent 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 library function `array_push` 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. **►Repetition?◄**

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. **►Is this all we can state about the objects?◄**

►Rewrite◄

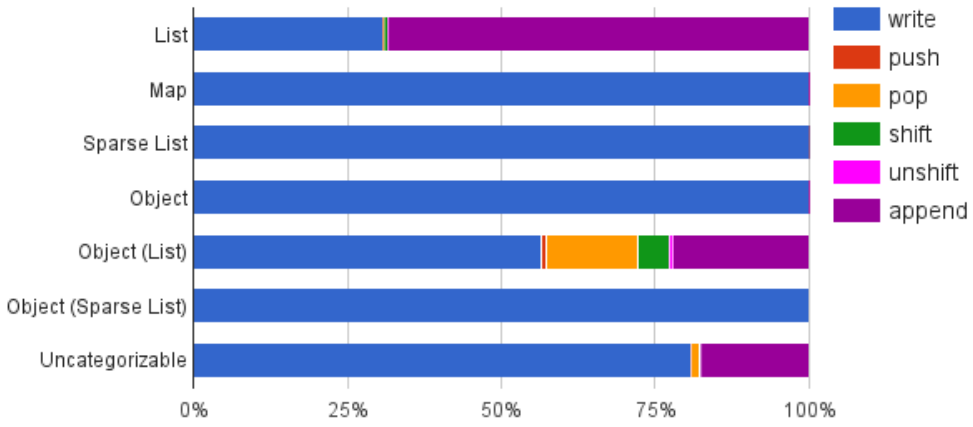


Figure 3.5: Distribution of array-changing operations over array types

3.4 Conclusion

The purpose of the dynamic analysis was to evaluate our hypothesis stating that arrays in a given program can be categorized as either list, map, or object, and once categorized, will stay in the category throughout multiple executions of that program. The results of the analysis supports this hypothesis by showing that arrays generally keep the same type. Furthermore, the usage of array

operations normally associated with lists, proved almost exclusive for arrays categorized as lists. This information should used to define unexpected behaviour reported by the static analysis, e.g. reporting `array_pop` preformed on a map.

The initial definitions categorized arrays as objects based on the type of the values. In the analysis however, a significant number of arrays turned out to be lists with different type values, defining them as objects. Operations on these objects were often list-operations, which indicates that categorizing based on value type might not be a meaningful strategy. It can be considered if the object type is providing enough significant information in itself, or should be consumed by the definitions of maps and lists, i.e. letting maps and lists allow values of different type.

Almost every framework in the corpus contains cyclic arrays however these are nearly exclusively uses of the superglobal `$GLOBALS`. Manipulating the global variables as a method for passing information seems suspicious and indicates poor program design, but reminds us that every PHP program has at least one cyclic array which must be handled in an sound manner. No creation of cyclic arrays has been detected in the program, which follows our hypothesis, that the developer will generally not utilize the more advanced array structures.

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 and adding elements of a different type to an array.

The definitions of maps and lists from chapter 3 are expanded to each absorb a part of the object type.

Definition 6. Let a be an array with integer keys 0 to $count(a) - 1$ defined, a is then considered an array-list.

Definition 7. Let a be an array with some integer key in the range 0 to $count(a) - 1$ is *not* defined, a is then considered an array-map.

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 performed using the monotone framework. Using 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) an instance of the monotone framework is derived in section 4.7. Since PHP relies heavily on coercion, a separate section (section 4.5) covers coercion of abstract values. Section 4.6 covers abstract evaluation and in section 4.8 the implementation of the analysis is briefly introduced.

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 and a limited number of statements but the language does support all reference features, e.g. reference assigning entries in an array. Initializing an array with references is not supported directly, however this can be done by initializing an empty array and writing or appending the initial values one by one. Reading from and writing to the `$GLOBALS` array will get and set the value of the variables, respectively. Creating a new variable by adding a 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 a program is only valid in P0 if it is also a valid PHP program. For example the syntax allows negation of arrays, however 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.

PHP supports using the array access syntax with strings to read and write individual characters of the given string. P0 does not support accessing strings with array syntax. The full PHP language defines a lot of cross-type operations however P0 supports only the operations described in sections 4.5 and 4.6.

►elaborate◄

►Write about support of alias◄

►Write about missing implementation of special expression: empty and the like◄

4.2 Control flow graph

The control-flow graph defines the basic instructions of a program and how data flows between those instructions. Nodes corresponds to instructions and edges to data flow. The graph is built recursively by defining sub graphs corresponding to grammar 4.1. Table 4.1 gives a description of sub graphs and their respective arguments.

For each sub graph with an argument this argument corresponds to the target argument (c_{tar}) of the last node in the sub graph. For instance let $E =$ `1+(2+3)` then the graph of $[\![E]\!](t)$ is recursively constructed as in graph 4.1. Notice how the argument, t, of the sub graph is the third argument of the last $bop_+(\_)$ node.

Definition 8. A control-flow graph is a four-tuple $G = (V, E, s, t)$, 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 of a control-flow graph is marked with an ingoing edge with no origin and the exit node is marked with an out-

⟨*program*⟩ ::= (⟨*function-definition*⟩ | ⟨*statement*⟩)*

⟨*function-definition*⟩ ::= '`function`' '`&`'?⟨*function-name*⟩ '`(`' ('`&`'? ⟨*var*⟩ ('`,`' '`&`'? ⟨*var*⟩)*) |
ϵ'`)`' ⟨*block*⟩

⟨*statement*⟩ ::= '`while (`' ⟨*expr*⟩ '`)`' ⟨*statement*⟩
| '`for (`' ⟨*expr*⟩?'`;`'⟨*expr*⟩?'`;`'⟨*expr*⟩? '`)`' ⟨*statement*⟩
| '`if(`'⟨*expr*⟩'`)`' ⟨*statement*⟩
| '`if(`'⟨*expr*⟩'`)`' ⟨*statement*⟩ '`else`' ⟨*statement*⟩
| '`;`'
| ⟨*expr*⟩'`;`'
| '`global`' ⟨*var*⟩ ('`,`' ⟨*var*⟩)*'`;`'
| '`return`' (⟨*expr*⟩|⟨*rexpr*⟩)?'`;`'
| ⟨*block*⟩

⟨*expr*⟩ ::= ⟨*expr*⟩ $\oplus$ ⟨*expr*⟩
| $\circ$ ⟨*expr*⟩
| '`(`' ⟨*expr*⟩ '`)`'
| ⟨*vexpr*⟩'`+ +`'
| ⟨*vexpr*⟩'`- -`'
| '`+ +`'⟨*vexpr*⟩
| '`- -`'⟨*vexpr*⟩
| ⟨*var*⟩
| ⟨*expr*⟩ '`[`'⟨*expr*⟩ '`]`'
| ⟨*function-reference*⟩
| ⟨*const*⟩
| ⟨*assignment*⟩
| '`[`' (ϵ | ⟨*array-init-entry*⟩ ('`,`' ⟨*array-init-entry*⟩)*) '`]`'
| '`array(`' (ϵ | ⟨*array-init-entry*⟩ ('`,`' ⟨*array-init-entry*⟩)*) '`)`'

⟨*function-reference*⟩ ::= ⟨*function-name*⟩'`(`' (⟨*function-arg*⟩ ('`,`' ⟨*function-arg*⟩)* | ϵ) '`)`'

⟨*function-arg*⟩ ::= ⟨*expr*⟩
| ⟨*rexpr*⟩

⟨*rexpr*⟩ ::= ⟨*var*⟩
| ⟨*function-reference*⟩
| ⟨*rexpr*⟩'`[]`'
| ⟨*rexpr*⟩'`[`' ⟨*expr*⟩ '`]`'

⟨*vexpr*⟩ ::= ⟨*var*⟩
| ⟨*rexpr*⟩'`[]`'
| ⟨*rexpr*⟩'`[`' ⟨*expr*⟩ '`]`'

⟨*array-init-entry*⟩ ::= ⟨*expr*⟩ '`=>`' ⟨*expr*⟩
| ⟨*expr*⟩

⟨*assignment*⟩ ::= ⟨*rexpr*⟩'`[]`' '`=`' ⟨*expr*⟩'`;`'
| ⟨*rexpr*⟩'`[`' ⟨*expr*⟩ '`]`' '`=`' ⟨*expr*⟩'`;`'
| ⟨*var*⟩ '`=`' ⟨*expr*⟩
| ⟨*rexpr*⟩'`[]`' '`=`' '`&`' ⟨*rexpr*⟩'`;`'
| ⟨*rexpr*⟩'`[`' ⟨*expr*⟩ '`]`' '`=`' '`&`' ⟨*rexpr*⟩'`;`'
| ⟨*var*⟩ '`=`' '`&`' ⟨*rexpr*⟩

⟨*block*⟩ ::= '`{`' ⟨*statement*⟩* '`}`'

Grammar 4.1: P0 syntax

Symbol	Description
$t \in \text{Temps}$	A temporary variable name used to pass evaluated values between CFG nodes
$h \in \text{HeapTemps}$	A temporary heap variable name used to pass a set of heap locations between CFG nodes
$c \in \text{Temps} \bigcup \text{HeapTemps}$	Either of the above two types of variable names
$[\![E]\!](t)$	Sub graph for expression E with evaluation result stored in t
$[\![R]\!](h)$	Sub graph for reference expression R with evaluated heap-location set stored in h
$[\![V]\!](h)$	Sub graph for a variable expression V with evaluated heap-location set stored in h
$[\![T]\!](c)$	Sub graph for either an expression, a reference expression or a variable expression with c being the corresponding t or h as described above
$[\![S]\!]$	Sub graph for statement S

Table 4.1: Symbols used in the control flow graph

going edge with no target. E.g. in example graph 4.1, $s = \mathit{constRead}(1, t_1)$ and $t = \mathit{bop}_+(t_1, t_2, t)$.

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.

$\mathit{bop}_\oplus(t_l, t_r, t_{tar})$, $\mathit{sop}_\oplus(t_l, t_r, t_{tar})$, $\mathit{uop}_\circ(t, t_{tar})$, $\mathit{inc}_\circ(h, t_{tar})$: These nodes indicates binary, short-circuit-binary, unary, and increment/decrement operations, respectively. The operations of the first three are all performed on temporary storage, while the fourth is performed on the heap. The last argument, t_{tar}, indicates where the result of performing the operation 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}. The arguments t_l and t_r corresponds to entries in the temporary storage containing the values of the left operand and the right operand respectively. For unary operations the t argument is where the value of the operand is stored in temporary storage and for the increment and decrement operations the h variable is the set of heap-locations storing possible operand values.

if(t): This node has one incoming and two outgoing edges, representing the choice of one branch or the other. The argument t is the entry in temporary storage where the evaluated value of the condition is stored.

constRead(k, t_{tar}): This node represents reading a constant into the temporary storage at t_{tar}. The constant k can be a string, a boolean, null or a number.

varRead$(\texttt{\$v}, c_{tar})$, *varWrite*$(\texttt{\$v}, c_{val}, t_{tar})$: These nodes indicate 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 variable name.

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

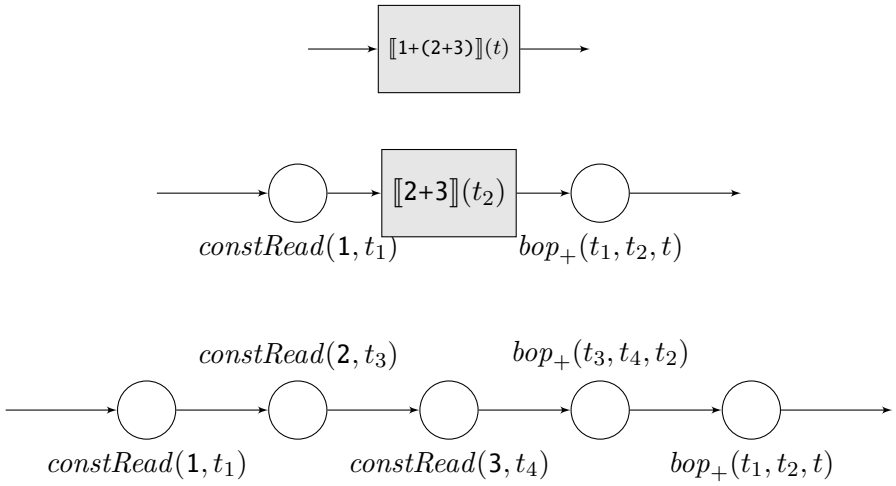
arrayAppend(t_{val}, t_{arr}), *arrayAppend*$(h_{var}, c_{val}, t_{tar})$, *arrayAppend*$(h_{var}, h_{tar})$: With three different signatures, this node represents appending to an array in temporary storage, in the heap, or in a read-context. Appending to an array in temporary storage happens when an array is initialized with values without key, in this case the first type of append node is used. Appending to an array in the heap is an ordinary array append where h_{var} is the possible locations of the array, c_{val} is either a heap temporary variable name in case of a reference append or a temporary variable name in case of a normal append. The value of c_{val} is stored in t_{tar}. Appending in a read context, i.e. without a value to be written, is usually not allowed however there are some exceptions. For those case the last node is used and the appended heap location is placed in temporary heap storage for later access.

arrayRead$(h_{arr}, t_{key}, c_{tar})$: Reading from an array on the heap (h_{arr}) with a key derived from the value at t_{key}. The result can either be a reference to the entry or the value 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 a reference or a value, hence the c_{val} variable. The value evaluated from c_{val} is stored in temporary storage at t_{tar}.

call$_{fn}(c_1, \cdots, c_n)$, *exit*$(c_1, \cdots, c_m)$, *result*$_c(c_{tar})$: A function call is performed with the *call* node. This holds the name, *fn*, of the function being called (which can always be resolved) as well as information about function parameters $(c_1, \cdots, c_n)$. For every call node, v, there is a single result node, *result*$_v$. This restores the calling context and stores the result of the function call in c_tar. The node immediately before the result node is an *exit* node, which is unique to the function being called and the last node of the function sub graph. The exit arguments $c_1, \cdots, c_m$ are entries in the temporary storage which contain the evaluations of return statements in 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.



Graph 4.1: Recursively constructing graph ⟦1+(2+3)⟧(t)

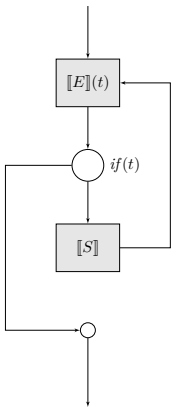
After parsing a P0 program $p : \langle program \rangle$ to an abstract syntax tree the control-flow graph, $G_p = (V_p, E_p, s_p, t_p)$, can be constructed recursively by constructing sub graphs for statements $S_1, \cdots, S_n$ in p. The sub graphs, $G_i = (V_i, E_i, s_i, t_i)$ and $G_{i+1} = (V_{i+1}, E_{i+1}, s_{i+1}, t_{i+1})$, for statement i and $i + 1$ respectively are connected by edges from t_i to s_{i+1}. A separate *start* node is created as s_p and $t_p = t_n$. For each function definition in p a separate graph, $G_f = (V_f, E_f, s_f, t_f)$, is created with $s_f = start_f$ and $t_f = exit_f$. Function calls in p create edges to the corresponding s_f and from the corresponding t_f.

4.2.1 Statements

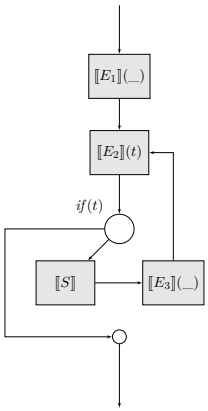
Let $s : \langle statement \rangle$ be a statement, then there are nine different graphs, one for each case in the grammar 4.1. The first four, for $s =$ `while(`E`)` S, $s =$ `for(`E_1`;`E_2`;`E_3`)` S, $s =$ `if(`E`)` S, and $s =$ `if(`E`)` S_1 `else` S_2 statements, are depicted as graph 4.2.1, 4.2.2,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

Return-statements, $s =$ `return` T, have three different graphs. An empty statement, i.e. it does not contain any expressions, is equivalent to returning `null`. This case yields the graph in 4.3.1. If $T : \langle rexpr \rangle$ 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 : HeapTemps). If this is not the case, (see program 4.1.1), and 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 is not a valid PHP program and thus not a valid P0 program (see section 4.1). Assuming $T : \langle expr \rangle$ the graphs for a return statement evaluates T and return the result. This is the case in graph 4.3.2 with c : Temps. For all graphs, the exit-node is the unique exit-node of the function.

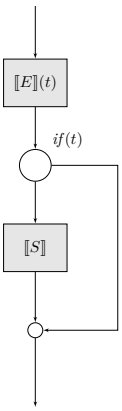
The remaining four graphs are the empty graph, the graph of the expres-



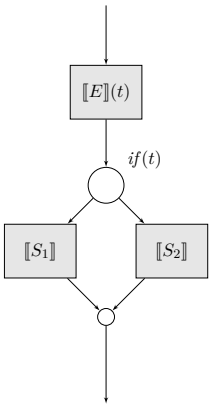
(4.2.1) **while**-statement



(4.2.2) **for**-statement



(4.2.3) **if**-statement



(4.2.4) **if-else**-statement

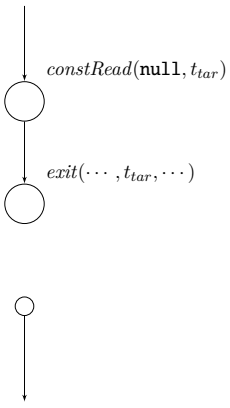
Program 4.1 Return-statement examples

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

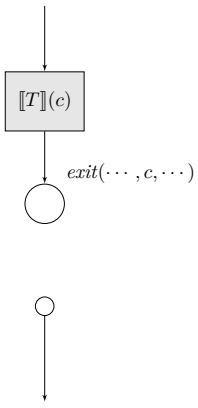
(4.1.1) Normal return-statement

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

(4.1.2) Reference return-statement



(4.3.1) **return**-statement

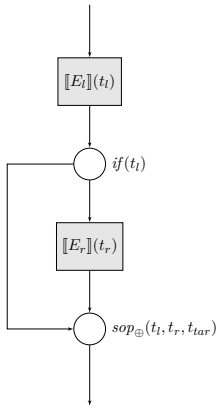


(4.3.2) **return**-expression-statement

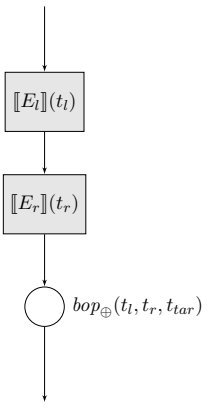
sion statement, the graph of `global` statement, and the graph of the block statement, which are all straight-forward.

4.2.2 Expressions

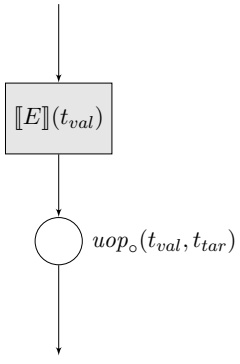
Let $e : \langle expr \rangle$, then by ignoring the trivial parenthesized expression case, viewing the four increment/decrement operations as one, and the two array-initialization operations as one, the expression can be on nine different forms. For $e = E_l \oplus E_r$, the graph depends on the operation. If the operation is a short-circuit operation, logical `&&` or `||`, then the graph is as 4.4.1, since 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, since 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 is 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.



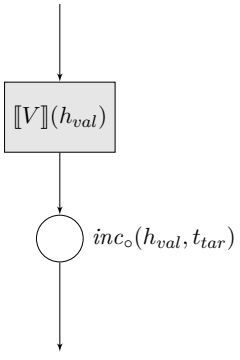
(4.4.1) Short-circuit binary operation



(4.4.2) Binary operation



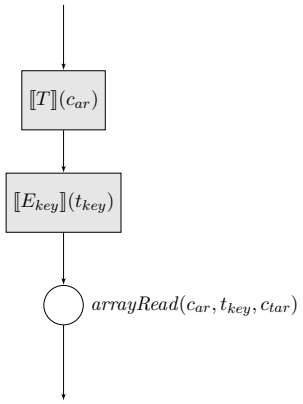
(4.4.3) Unary operation



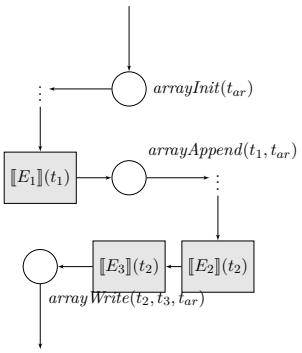
(4.4.4) Post/pre increment/decrement opreation

For a variable read $e =$ `$v` (for some variable `$v`), or a constant read, e.g. $e =$ `"foo"`, the graph is a single $varRead(\texttt{\$v}, t_{tar})$ or $constRead(\texttt{"foo"}, t_{tar})$ node, respectively. For an array read expression, $e = E_{ar}\texttt{[}E_{key}\texttt{]}$, 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_{arr} and c_{arr}, c_{tar} : Temps. If the expression is an array initialization, $e =$ `[`$\cdots, E_1, \cdots, E_2$`=>`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 $e = fn\texttt{(}T_1, \cdots, T_n\texttt{)}$, the result variable is a temporary variable, c_{tar} : Temps, the arguments are either an expression, $T_i : \langle expr \rangle$



(4.5.1) Array read



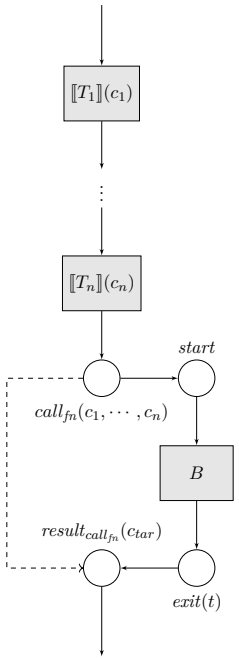
(4.5.2) Array initialize

and c_i : Temps, or a reference expression, T_i : $\langle expr \rangle$ and c_i : 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, the argument is an expression. This follows from the program being a valid P0 program. The function graph will be as graph 4.6, 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 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, used for restoring the local context before calling *fn*.

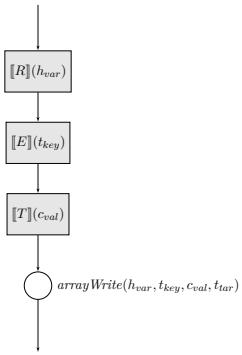
Finally, the expression can be an assignment. There are two types of assignments, a regular value-assignment and a reference-assignment, using the `&` operator. Each assignment can be split up in three categories; variable-, array-write-, and array-append-assignments, depending on the target (left-hand 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} : 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} : $\langle expr \rangle$ and c_{val} : HeapTemp the heap temporary variable holding the heap locations of the value.

4.2.3 Reference and variable expressions

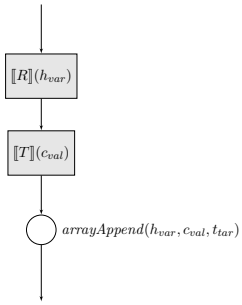
Since PHP supports nested assignments, e.g. `$a['foo'] = &func()[]`, deciding which location to update is performed recursively. The graph of this assignment is illustrated as graph 4.8. Here the locations of variable `$a` is



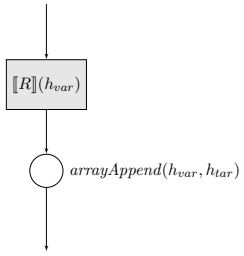
Graph 4.6: Function call



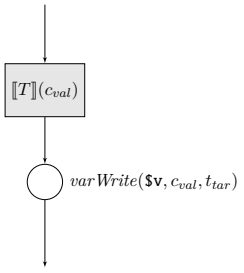
(4.7.1) Array write



(4.7.2) Array append



(4.7.3) Array append without write

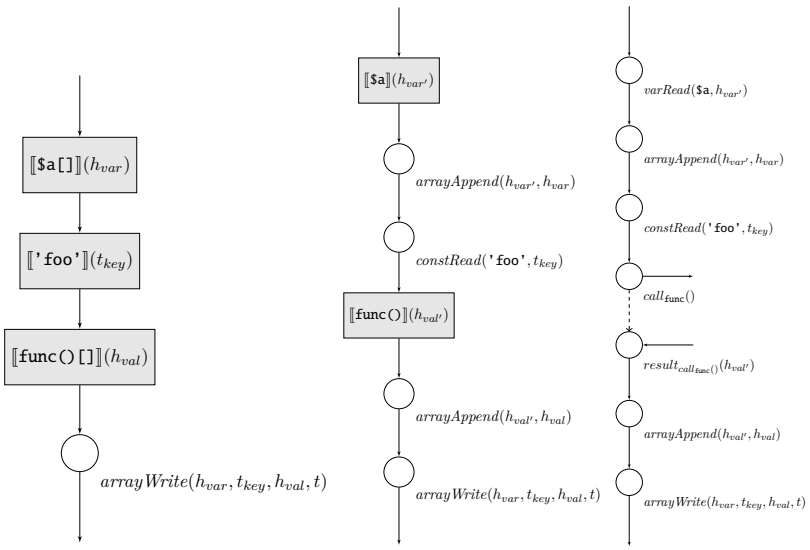


(4.7.4) Variable write

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
```

stored in variable $h_{var'}$. A new location l is then 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-hand side are resolved and stored 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 sub trees which takes a h : 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-location sets rather than values. For $r = fn(c_1, \cdots, c_n)$ the graph corresponds to graph 4.6, with a heap-location set as the result, i.e. c_{tar} : HeapTemps. For the variable read $r =$ `$v` the graph is a single $varRead(\texttt{\$v}, h_{tar})$ node. When the expression is an array-read $r = R_{arr}[E_{key}]$ the graph corresponds to that of 4.5.1 with c_{arr}, c_{tar} : HeapTemps and T being the graph of R_{arr}. Finally, for the append operation $r = R_{arr}[]$ the graph corresponds to that of graph 4.7.3.

4.3 Lattice

The lattice is the data-structure passed around by the flow of the control-flow graph. In order to create a *inter-procedural analysis*, the analysis lattice is defined as

$$\text{AnalysisLattice} = \Delta \rightarrow \text{State} \tag{4.1}$$

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 resulting in a unsound analysis. In the analysis the only strong update of the heap is performed when writing to a variable pointing to no locations which is sound, since the variable has not been initialized and hence cannot have been referenced. By only using the global scope, performing a strong update with a not-null value will indicate that the variable cannot 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.

$$\Delta = \text{CallNode}_*^{\leq k} \tag{4.2}$$

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

$$\text{State} = \text{Locals} \times \text{Globals} \times \text{Heap} \times \text{Temps} \times \text{HeapTemps} \tag{4.3}$$

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, since global variables can always be accessed from a function using the `global` statement. Furthermore, the super-global variables reside in the global scope but can always be accessed directly. Two scopes are enough, because any other variables have to be passed to a function as an argument.

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

The scopes are defined as maps from variable names in Var to power-sets 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 the feature of assigning references to and from variables and array entries. This is done using the `&` operator and makes the heap abstraction necessary. The heap allows values to be used by multiple variables and arrays, which enables proper propagation of changes.

$$\text{Heap} = \text{HLoc} \to \text{Value} \tag{4.5}$$

The Temps and HeapTemps map-lattices store intermediate results. Since these results cannot be referenced, there is no need to store them in the heap. By keeping them in a separate lattice, they can be strongly updated and do 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 $\bot$ elements, which is avoided by this method.

$$\text{Temps} = \text{TVar} \to \text{Value} \tag{4.6}$$

$$\text{HeapTemps} = \text{THVar} \to \mathcal{P}(\text{HLoc}) \tag{4.7}$$

The necessity of the HeapTemps 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 variable names, TVar and THVar, are both finite following from the control-flow-graph being finite.

The heap locations are allocation site abstractions with respect to the context, node, and a natural number. The natural number allows the creation of multiple location per node, which is necessary in *call*-nodes to support multiple arguments. Adding the natural number as a factor makes the set of allocation sites possibly infinite, in practise however the set is finite since the number of function parameters is finite.

$$\text{HLoc} = \Delta \times \mathcal{N} \times \mathbb{N} \tag{4.8}$$

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

An abstract value is defined as the product lattice of the five factors defined by the Hasse diagrams shown in figure 4.1. These lattices were 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.

$$\text{Value} = \text{Array} \times \text{String} \times \text{Number} \times \text{Boolean} \times \text{Null} \tag{4.9}$$

Following the results of the dynamic analysis in chapter 3, the array is considered either a set of locations or a map from indices to sets of types, i.e. array-lists or array-maps. The sum-lattice has been chosen as opposed to a product-lattice, since the dynamic analysis indicated that arrays seldom change 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.

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

$$\text{ArrayMap} = \text{Index} \to \mathcal{P}(L) \tag{4.11}$$

The indices of the map-array-lattice is an infinite lattice, yielding a possibly infinite array-lattice. To ensure termination of the analysis the lattice must be finite, so it must be argued that the lattice is finite in practice. Assuming that an infinite-sized array exists, an infinite number of writes to a map is required. This in turn requires an infinite-sized program, a recursive function, or a loop. Since an infinite program is not possible one of the two latter cases must hold. Assuming the cause is a recursive function, the array must then be finite because the number of contexts are bounded and the number of heap locations are bounded. 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.

$$\text{Index} = \text{String} + \text{Integer} \tag{4.12}$$

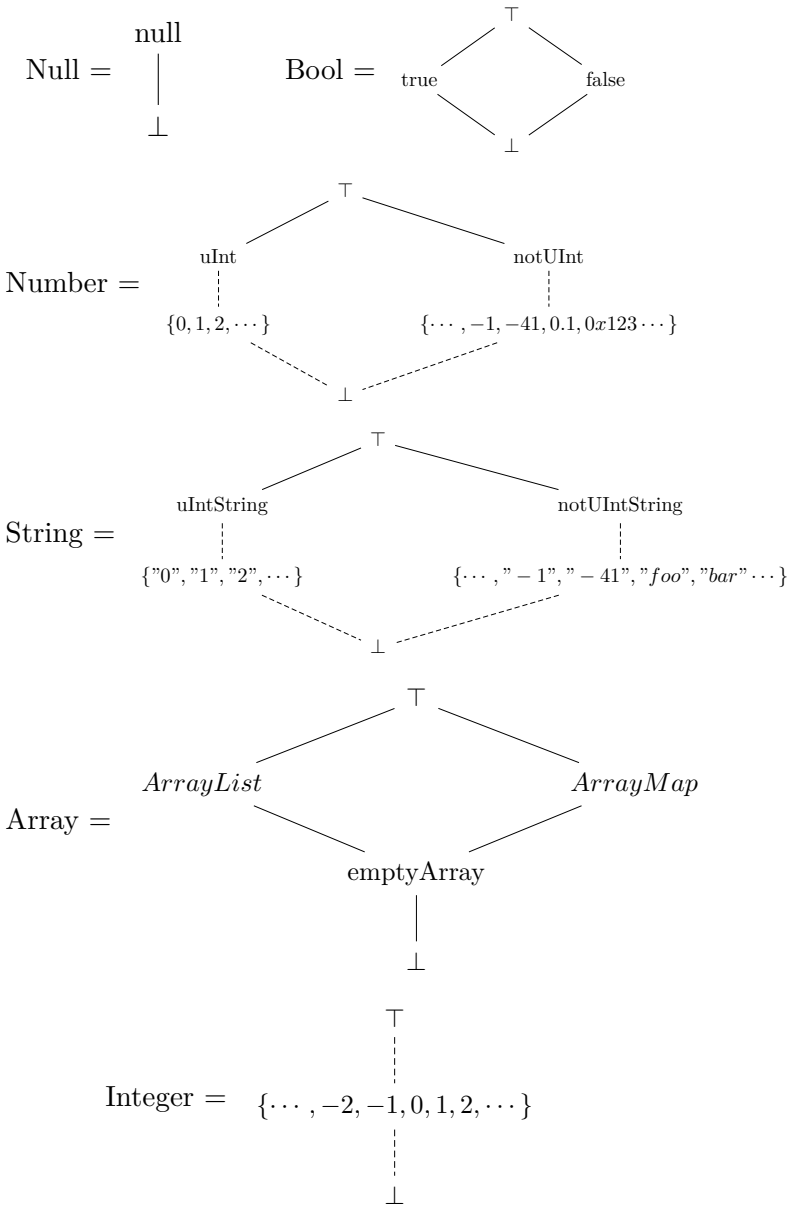


Diagram 4.1: Hasse diagrams of lattices

4.4 Transfer functions

To propagate the lattice-elements between nodes in the control-flow graph, each node type has a corresponding transfer function. Most of the transfer functions are defined on State instead of AnalysisLattice, i.e they are defined as $f_{n,\delta}$: State $\rightarrow$ State rather than $f_{n,\delta}$: AnalysisLattice $\rightarrow$ AnalysisLattice. This eases the notation. Given a state-transfer-function $f'_{n,\delta}$ the corresponding lattice-transfer function, $f_{n,\delta}$, can be defined as

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

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 in sections 4.4.1 through 4.4.5. In the definitions the 5-tuple $(s_l, s_g, s_h, s_t, s_{ht})$ is used to represent the five lattice-elements inside the State lattice-element given as input to the function.

4.4.1 Operations

Let $n = bop_{\oplus}(t_l, t_r, t_{tar})$ or $n = sop_{\oplus}(t_l, t_r, t_{tar})$. Then

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

Let $n = uop_{\circ}(t_{val}, t_{tar})$. Then

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

The soundness of the binary, unary, and short-circuit operations follow 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 $updateLocations : \mathcal{P}(\text{HLoc}) \times \text{Heap} \times (\text{HLoc} \rightarrow \text{Value}) \rightarrow \text{Heap}$ function writes a value to the heap using weak updates.

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

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

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

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

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

With a separate Locals and Globals scope, the current scope (with respect to a variable v) is decided by looking at the current context as well as 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 superglobal is done by the predicate *isSuperGlobal* which is true if and only if v is either `$_GET`, `$_POST`, `$_SESSION`, `$_COOKIE`, `$_SERVER`, `$_REQUEST`, `$_FILES`, `$_ENV`, or `$GLOBALS`.

An example of how the scope is decided can be seen in program 4.3. The current scope of the variable `$a` on line 4 in a function body is the local scope. The variable `$GLOBALS` on line 5 is a superglobal, and so is in the global scope. The variable `$c` is not in a function body, so it is in the global scope. The variable `$b` on line 3 is also in the local scope. This follows from the fact that 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 affect 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 $n = varWrite(v, t_{val}, t_{tar})$, the transfer function is defined as

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

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

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

In the latter case the variable is always strongly updated. This is sound, since the current language subset of PHP offers no ambiguity with respect to which variable is currently being updated, specifically because the infamous variable-variable feature is not supported.

Besides resolving the scope as above, reading a variable is quite straight forward. In order to be sound, however, the transfer function needs to take uninitialized variables into account. When reading an uninitialized variable in PHP, the default value is `NULL`. Because of this, if the read is stored in a temporary variable and the variable is not set in the current scope, the result should be Value(Null($\top$)). Otherwise the result should be the joined value of the heap locations pointed to by the variable. So for $n = varRead(v, t_{tar})$, the

transfer function is defined as

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

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 $\mathit{initializeVariable}_{n,\delta} : \mathrm{Var} \times \mathrm{Scope} \to \mathrm{Scope}$ function, which creates a new location in the provided scope, if none exists.

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

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

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

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 $n = \mathit{arrayInit}(t_{tar})$ the transfer function becomes

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

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

Array initialization

Array initialization happens when writing the expression `[···]` or `array(···)`, 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 an *arrayInit* node in the control-flow graph. 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 storage to an array likewise stored in the temporary storage is performed by first storing the value in the heap at some location l. Then the array is joined with a list-array containing only l. While this implies that an append operation on a map-array results in the $\top$-element 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. Furthermore, cases where the result is the $\top$-element array can be reported to the user as suspicious use. For $n = arrayAppend(t_{val}, t_{arr})$ the transfer function is defined in 4.25. The 5-tupel $(v_a, v_s, v_n, v_b, v_u)$ unfolds the Value lattice-element returned from the look-up in the temporary storage s_t.

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

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 an ordinary 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 using a newly created list-array lattice-element to join with. Here however the value being appended is also added to the temporary variable t_{tar}, since this is evaluation of the append-expression, e.g. `echo $a[] = 42;` prints the number `42`.

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

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

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

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

The final array-append operation occurs when an array is appended and immediately thereafter accessed, e.g. `$a['key']` where a location is appended to the array at `$a` and immediately thereafter implicitly initialized as an array 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 Value(Null($\top$)), it is important and sound to set l' to Value($\bot$), since the location will be joined with another array immediately after.

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

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

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 $c_{\mathrm{Value,ArrayIndex}}(v)$ introduced in section 4.5 directly, which coerces and then joins all factors. This approach would for instance coerce the value $v = (\bot, \texttt{"foo"}, 4, \bot, \bot)$ to the index $i = c_{\mathrm{Array,ArrayIndex}}(\bot) \sqcup \cdots \sqcup c_{\mathrm{Null,ArrayIndex}}(\bot) = \texttt{"foo"} \sqcup 4 = \top$.

Another approach would be to consider a set of possible indices I rather than a single index, by coercing the factors individually and accessing the array with each index. For the previous example v would yield the set $I = \{\bot, \texttt{"foo"}, 4\}$. Since the value will most likely contain at least one $\bot$ factor, the set of indices will also contain the $\bot$ array index, which will become a problem. This becomes apparent when first considering how values are 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. $a[\forall d \in \mathit{dom}(a) \wedge i \sqsubseteq d.d \mapsto a(d) \sqcup L]$. 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 $d \in \mathit{dom}(a)$ where $d \sqsubseteq i' \vee i' \sqsubseteq d$, which is practically feasible.

Returning to the problematic $\bot$ factors, since most writes would contain a at least one $\bot$ factor, most writes would, with the second approach, write to the $\bot$ index, and since $\bot$ is contained in all indices, massive loss of precision occurs. 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 $I = \{\texttt{"foo"}, 4\}$. These are the indices returned by the function

$$\mathit{indices} : \text{Value} \to \text{ArrayIndex}^*$$

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

Using the above function we are able to generalize reading from an array as a function $\mathit{readArray} : \text{Value} \times \text{Value} \times \text{Heap} \to \mathcal{P}(L)$ 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 about list indices. Reading from $\bot$ or emptyMap results in the empty set, since those 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.

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

In the same manner, the act of writing to an array can be generalized to the function $\mathit{writeArray} : \text{Value} \times \text{Value} \times \mathcal{P}(L) \to \text{Value}$. 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 any of the indices are strings, then a map is returned otherwise a list is returned. The predicate $\mathit{isInteger}(i)$ holds iff i is an

integer.

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

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. the right-hand side of the assignment `$b = $a['key']`, the node is $n = \mathit{arrayRead}(t_{arr}, t_{key}, t_{tar})$ where the value read from the array should be stored at t_{tar}. The query is always joined with Value(Null($\top$)) since an entry might not be set. The transfer function is defined as

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

Another array-read operation is 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 $n = \mathit{arrayRead}(h_{var}, t_{key}, h_{tar})$ and the transfer function becomes

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

where

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

All explicit array initializations initializes an empty array. Any initial data is added by subsequent array-appends (covered the section about array append) and array-write operations. For example initializing an array `['a'=>1, 'b'=>2]` two array-write operations follow, one for each entry. The transfer function for the corresponding control-flow node $n = \mathit{arrayWrite}(t_{key}, t_{val}, t_{arr})$ uses 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

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

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

as

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

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 $n = \mathit{arrayWrite}(h_{var}, t_{key}, h_{val}, t_{tar})$, where the transfer function is defined as

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

Notice that 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

or 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 become lists 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 calls (*call* and *result* nodes) differ from the other functions. For $n = call_{fn}(c_1, \ldots, c_n)$, the transfer function $f_{n,\delta}$: AnalysisLattice $\rightarrow$ 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 whether the call arguments, $c_1, \ldots, c_n$, are variable names in THVar or TVar.

If the argument is passed by reference, 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 turn 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 state, while the temporary

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

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

where the function $\mathit{addCallNode} : \Delta \times \mathrm{CallNode} \to \Delta$ decides the target context from the current, the function $\mathit{args} : \mathrm{FunctionNames} \to (\mathrm{Var} \times \mathbb{N})^*$ given a function name returns a list of arguments expressed as pairs of variable names and positions, and the function $\mathit{startNode} : \mathrm{FunctionNames} \to \mathrm{StartNode}$ 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 for higher efficiency.

After execution of a function it is the task of the transfer function of the result node, to restore the old execution context. For $n = \mathit{result}_{\mathit{call}_{fn}}(\_)$, the transfer functions are defined as functions from two lattice-elements to a single lattice-element: $f_{n,\delta_{call},\delta_{exit}} : \mathrm{AnalysisLattice} \times \mathrm{AnalysisLattice} \to \mathrm{AnalysisLattice}$, where the first lattice-element is the element passed to the transfer function of the call node call_{fn} and δ_{call} is the original context. The second lattice-element is the usual element derived from incoming flow. Depending on the argument of the *result* node, the transfer function is defined as either of 4.39 and 4.40

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

let $n = result_{call_{fn}}(t_{val})$ or $n = result_{call_{fn}}(h_{val})$ as

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

where the *exitNode* : FunctionName $\rightarrow$ ExitNode function given a function name returns the corresponding unique exit node. These functions will return a lattice-element 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 pass-by-reference respectively. In the latter case the position of the *exit-argument* is again used to decide which heap location to save the value at, 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-element in practice are finite, so is the number of heap-locations.

4.4.5 Other transfer functions

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

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

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

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

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

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 dynamically typed language, PHP supports coercion from most types to scalars (strings, numbers and booleans). Coercion is primarily used when performing binary and unary operations, and when accessing arrays. Coercion to an array index must be treated as a separate case, since it behaves differently from string or number coercion. For example 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 `$key = 4.34E1`, since keys must be either integers or strings. Accessing with `$key = "42.1"` accesses the entry with string key `"42.1"`.

Since there is no formal language specification, these coercion rules have 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-concatenation `.`-operator.

Coercion on abstract values can be performed using the $c_{\alpha,\beta} : \alpha \to \beta$ 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 abstract values is defined by the following functions

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

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

$$c_{\text{Array,String}}(v) = \begin{cases} \texttt{"Array"} & \text{if } v \neq \bot \\ \bot & \text{else} \end{cases} \qquad c_{\text{Null,String}}(v) = \begin{cases} \text{""} & \text{if } v = \top \\ \bot & \text{else} \end{cases}$$

$$c_{\text{Number,String}}(v) = \begin{cases} \text{uIntStr} & \text{if } v = \text{uInt} \\ \text{notUIntStr} & \text{if } v = \text{notUInt} \\ \top & \text{if } v = \top \\ \bot & \text{if } v = \bot \\ \mathit{string}(v) & \text{else} \end{cases} \qquad c_{\text{Bool,String}}(v) = \begin{cases} \texttt{""} & \text{if } v = \text{false} \\ \texttt{"1"} & \text{if } v = \text{true} \\ v & \text{else} \end{cases}$$

$$c_{\text{Array,Bool}}(v) = \begin{cases} \text{false} & \text{if } v = \text{emptyArray} \\ \bot & \text{if } v = \bot \\ \top & \text{else} \end{cases} \qquad c_{\text{Null,Bool}}(v) = \begin{cases} \text{false} & \text{if } v = \top \\ \bot & \text{else} \end{cases}$$

$$c_{\text{Number,Bool}}(v) = \begin{cases} \text{false} & \text{if } v = 0 \\ \bot & \text{if } v = \bot \\ \top & \text{if } v = \text{uInt} \vee v = \top \\ \text{true} & \text{else} \end{cases} \qquad c_{\text{String,Bool}}(v) = \begin{cases} \text{false} & \text{if } v = \texttt{""} \vee v = \texttt{"0"} \\ \bot & \text{if } v = \bot \\ \top & \text{if } v = \text{uIntString} \vee v = \top \\ \text{true} & \text{else} \end{cases}$$

$$c_{\text{Null,Number}}(v) = \begin{cases} 0 & \text{if } v = \top \\ \bot & \text{else} \end{cases} \qquad c_{\text{Array,Number}}(v) = \bot$$

$$c_{\text{Bool,Number}}(v) = \begin{cases} 1 & \text{if } v = \text{true} \\ 0 & \text{if } v = \text{false} \\ \text{uInt} & \text{if } v = \top \\ \bot & \text{if } v = \bot \end{cases} \qquad c_{\text{String,Number}}(v) = \begin{cases} \mathit{num}(v) & \text{if } \mathit{isNumber}(v) \\ \text{uInt} & \text{if } v = \text{uIntString} \\ \top & \text{if } v = \text{notUIntString} \\ \top & \text{if } v = \top \\ \bot & \text{if } v = \bot \\ 0 & \text{else} \end{cases}$$

$$c_{\text{Array,Index}}(v) = \bot \qquad c_{\text{Bool,Index}}(v) = c_{\text{Bool,Number}}(v)$$

$$c_{\text{Number,Index}}(v) = \begin{cases} \mathit{int}(v) & \text{if } \mathit{isNumber}(v) \\ \bot & \text{if } v = \bot \\ \top & \text{else} \end{cases} \qquad c_{\text{String,Index}}(v) = \begin{cases} \mathit{int}(v) & \text{if } \mathit{isInteger}(v) \\ v & \text{if } \mathit{isString}(v) \\ \top & \text{else} \end{cases}$$

$$c_{\text{Null,Index}}(v) = \begin{cases} \text{""} & \text{if } v = \top \\ \bot & \text{else} \end{cases} \qquad c_{\alpha,\alpha}(v) = v$$

The functions *int*, *num*, and *string* creates an integer, a number, and a string respectively in the obvious way, e.g. $\mathit{int}(42.1) = 42$, $\mathit{string}(1337) = \texttt{"1337"}$, and $\mathit{number}(\texttt{"1337"}) = 1337$. Furthermore the predicates *isNumber*, *isInteger* and *isString* are satisfied if and only if the value can be interpreted as a number, integer, or string respectively. For example $\mathit{isNumber}(\texttt{"42.1"})$ is satisfied while $\mathit{isInteger}(\texttt{"42.1"})$ does not.

Five interesting cases of coercion in PHP includes: (i) any array coerced to a string will result in the string `"Array"`, (ii) `null` is string-coerced to the empty string, (iii) both the empty string and the string literal `"0"` are boolean-coerced to boolean `false`, (iv) boolean `false` is string-coerced to the empty string, and (v) an empty array is boolean-coerced to boolean `false`.

4.6 Abstract evaluation

The PHP language syntax contains a set of binary and unary operators. These operators evaluate one or two operand values to a result value. Working with abstract values introduces the need for a abstract evaluation of these operators. Being dynamically typed PHP allows for many cross-type operations to be performed some of which simply coerces the operands to sensible types. However some exceptions exists, e.g. performing the increment operation on strings. P0

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

Table 4.2: Binary operators

does not support these exceptions. The operations are supported on the types on which they are defined, and cross-type operations are supported by using the coercion functions defined in the previous section.

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

The binary and unary operators supported are listed in table 4.2 and 4.3 respectively. Associated with them is their signature, indicating on which values they are defined. The operators are generalized to values by using the coercion

Operator	Name	Signature
`! x`	Negation	Boolean → Boolean
`- - x`	Pre-decrement	Number → Number
`x - -`	Post-decrement	
`+ + x`	Pre-increment	
`x + +`	Post-increment	
`- x`	Unary Minus	

Table 4.3: Unary operators

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

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

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

In table 4.2 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. The variables x and y denote an integer larger than zero ($x, y \in \mathbb{Z} \wedge x, y > 0$) and the variables a, b denotes a double smaller than zero or not an integer, $a, b \in \mathbb{R} \wedge (a, b < 0 \vee a, b \notin \mathbb{Z})$. Rephrased; x and y are uInt numbers and a and b are notUInt numbers, e.g. with this notion the addition $x + a$ would be an addition of a uInt number (x) and a notUInt number (a).

Table 4.4 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. The right-identity of subtraction is 0 which is used in the 0-column of table 4.4 to get uInt instead of $\top$. Subtracting negative numbers correspond to adding the absolute value of the number, e.g. $5 - (-3) = 5 + 3$. For uInt this means that adding negative integers results in uInt instead of $\top$.

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

Table 4.4: Abstract 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 instead of 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 4.5 the amount of possible `false` values is high for the modulus operator. This is because PHP handles modulus for decimal numbers by truncating 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.

%	$\bot$	0	y	uInt	$1 > b > -1$	b	notUInt	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	f	0	$0 \sqcup f$	$0 \sqcup f$	0	$0 \sqcup f$	$0 \sqcup f$
x	$\bot$	f	$x\%y$	$\text{uInt} \sqcup f$	$x\%b$	$x\%b$	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$
uInt	$\bot$	f	uInt	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$	uInt	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$
$1 > a > -1$	$\bot$	f	$a\%y$	$\text{uInt} \sqcup f$	$a\%b$	$a\%b$	$\text{uInt} \sqcup f$	$\text{uInt} \sqcup f$
a	$\bot$	f	$a\%y$	$\text{notUInt} \sqcup f$	$a\%b$	$a\%b$	$\text{notUInt} \sqcup f$	$\text{notUInt} \sqcup f$
notUInt	$\bot$	f	$\top$	$\top \sqcup f$	$\top \sqcup f$	$\top$	$\top \sqcup f$	$\top \sqcup f$
$\top$	$\bot$	f	$\top$	$\top \sqcup f$	$\top \sqcup f$	$\top$	$\top \sqcup f$	$\top \sqcup f$

Table 4.5: 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 4.6. Any combination of operators not present in the table are considered unspecified and yields a boolean $\top$ value.

For performance reasons, the analysis does not try to compare all different combinations of possible types when performing abstract comparison operators. If either value has multiple possible types the result is the boolean $\top$-element. Otherwise the comparison operators follow the order of table 4.6 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, while arrays of equal size are compared by their values. Since the array lattice has 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 4.6: 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 in [**?**], is used. This section describes how the concepts of the previous sections fits into the framework. 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, a 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 flow 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 $A = \mathcal{L} \to L$ are defined as

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

where

$$A_\circ(l) = \bigsqcup\{A_\bullet(l') | (l', l) \in F\} \sqcup \iota^l_{E'} \tag{4.44}$$

and

$$\iota^l_{E'} = \begin{cases} \iota & \text{if } l \in E' \\ \bot & \text{else} \end{cases} \tag{4.45}$$

Intuitively, $A_\bullet$ expresses the abstract state immediately after performing the given label and $A_\circ$ the state just before. Solving these equations performs the data-flow analysis.

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

$$\begin{aligned} F = \{&((n,\delta),(n',\delta')) | (n,n') \in E \wedge \delta \in \Delta \\ &\wedge \mathit{validSuccessor}(n,n') \\ &\wedge \delta' = \mathit{nextC}(n,\delta)\} \end{aligned} \tag{4.46}$$

where

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

The predicate *validSuccessor* is defined by definition 9, and it is necessary because of the ambiguity associated with potentially multiple outgoing edges of the *exit* node, which indicates that the flow may go from an exit node to an arbitrary result node. This is naturally not the case. A successor w to a node v is valid if and only if 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 9. The predicate $validSuccessor : \mathcal{L} \times \mathcal{L}$, is defined as

$$\begin{aligned} validSuccessor(n, \delta, n', \delta') \Leftrightarrow & (n = exit(\_) \wedge n' = result_c(\_) \wedge \delta = (\delta' c)) \\ & \vee n \neq exit(\_) \end{aligned} \tag{4.48}$$

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

$$\begin{aligned} g_\iota = [&\texttt{\$GLOBALS} \mapsto \{\text{HLoc}(\Lambda, s, 0)\}, \\ &\texttt{\$\_SERVER} \mapsto \{\text{HLoc}(\Lambda, s, 1)\}, \\ &\texttt{\$\_SESSION} \mapsto \{\text{HLoc}(\Lambda, s, 2)\}, \\ &\texttt{\$\_ENV} \mapsto \{\text{HLoc}(\Lambda, s, 3)\}, \\ &\texttt{\$\_COOKIE} \mapsto \{\text{HLoc}(\Lambda, s, 4)\}, \\ &\texttt{\$\_POST} \mapsto \{\text{HLoc}(\Lambda, s, 5)\}, \\ &\texttt{\$\_GET} \mapsto \{\text{HLoc}(\Lambda, s, 6)\}, \\ &\texttt{\$\_REQUEST} \mapsto \{\text{HLoc}(\Lambda, s, 7)\}, \\ &\texttt{\$\_FILES} \mapsto \{\text{HLoc}(\Lambda, s, 8)\}] \end{aligned}$$

and the heap, h_ι is

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

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 equations of the monotone framework, the above lattice, control-flow-graph, and transfer functions have been implemented in approximately 8100 lines of Java code as a plug-in for the Intellij IDEA (Ultimate edition) development environment by JetBrains[1]. This IDE supports multiple

[1] `http://jetbrains.com`

languages, such as Java, Python, C, C++, C#, Ruby, and PHP with tools such as re-factoring and type-checking, and a vast library of plug-ins, developed by JetBrains or the JetBrains community. The full source code of the implementation can be found at `http://github.com/Silwing/tapas`.

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 in a single pass of these elements, where each node keeps a reference to the element from which it was 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 Value(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* also ensures 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 $\iota \not\sqsubseteq \bot$, 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 to 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

Algorithm 1 Worklist algorithm

Require: Control flow graph, $G = (V, E, s, t)$
1: $I = [\forall \delta \in \Delta, n \in \mathcal{N}.(n, \delta) \mapsto \bot]$
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**

reference or value, and specifying a transfer function for the result node. 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 =`array_merge`[2] the arguments and return value are all passed by value. The transfer function for the result node, $n = result_c(c_{tar})$, where $c = call_{fn}(t_1, \cdots, t_n)$, is defined as

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

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

[2] `http://php.net/manual/en/function.array-merge.php`

these are not representative of actual PHP programs, neither in size nor feature usage, they do aim to cover as many feature cases as possible.

►**Elaborate on more library functions.**◄

4.9 Summary

In this chapter a subset of PHP (P0) has been defined. This language focuses on arrays by supporting native array operations, such as array append, write, and initialization as well as references. References are used to create more advanced arrays, e.g. cyclic arrays, and has proven as a source of precision loss, since they disable deep-copy of arrays, and in turn strong updates on the heap.

An interprocedural data-flow analysis has been designed in the monotone framework, yielding a method for creating control-flow-graphs, a lattice expressing abstract state, and a set of transfer-functions. By relying on the monotone framework, the analysis has been performed using a straight-forward implementation of the worklist-algorithm as a plug-in in the development environment Intellij IDEA.

►**write about soundness and monotonicity**◄

Chapter 5

Case Study

Evaluating the analysis presented in chapter 4 is performed by analysing six small P0 programs. The first program is a constructed example illustrating how the analysis handle the two introduced array types. The last five programs have been found by manually inspecting the corpus of frameworks used in chapter 3, looking for uncategorizable arrays and misuse of list-operations, e.g. `array_pop` on a map. Discovering these programs did prove more challenging than first anticipated, partially because many of the uncategorizable arrays proved to be multidimensional arrays, which can be misclassified, as discussed in section 3.2.5. When found, the programs were minimized (i.e. unnecessary logic were removed), rewritten to P0, and necessary library functions implemented. Running the analysis was performed with a context bound by length two, $k = 2$, which imposes no practical restriction on the analysis►**not true, 5.3 uses recursion**◄, since the maximum possible depth of function calls in the examples is two. Since implementation of a fast and resource efficient analysis isn't a goal of this thesis, execution statistics, such as running time and memory consumption, has been omitted.

5.1 Month names and number

Program 5.1 first define an list-array of month names, `$monthNames`, and use a loop to create a new map-array, `$monthMap`, with these month names as keys and the corresponding month number as value, e.g. given `$monthNames = ["January", ... , "December"]` after running the program the map becomes `$monthMap=["January"=>1, ... , "December" => 12]`. The constructed map-array is then used to print a string containing month number of a given input, e.g. the input `"August"` yields the string `"August is month number 8"` and input `"Duck"` yields `"Duck is month number "`. The input is provided via. the super-global `$_GET`, containing the data of the query string[1], as field `"monthName"`.

The array creation at line 1 is preformed by initializing an empty array and immediately thereafter appending the month names. With the only operation performed on the array being appends, the analysis will represent the array as a

[1] `http://tools.ietf.org/html/rfc3986#section-3.4`

Program 5.1 Month name and number example

```
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];
```

list of strings. At line 2 the `$monthMap` variable is initialized to an empty array. The analysis will represent this as emptyArray, since no operations are yet performed on the array. Notice how this explicit initialization isn't necessary since any subsequent array operation will implicitly initialize the array. The month map array is first modified at line 5 with an array-write operation with keys from the month-names array. Since the function `array_pop` is used to generate the key, which may return `null` if the given array is empty, `$monthMap` will be represented as a map-array with NotUIntString mapping to UInt.

The analysis yields no error or warning, following the arrays being treated properly. I.e. when first represented as either lists or maps only the *right* operations are performed on them, e.g. writes with string keys on maps and `array_pop` on list.

5.2 Array pivot

The function, `pivot`, in program 5.2, was found in the Joomla content management system, in a class containing functions for manipulating arrays. It uses the library function `count`, which returns the size of a given array, and the special `empty` expression, which has been replaced by the function `is_empty` that takes a single argument and returns a boolean.

The `pivot` function accepts an array as its only argument and return an array, with the unique values of the input array as keys and their position in the original array as value. E.g. executing `pivot([1,2,3,4,5,4,3,2,1])` yields the map

```
[
  1 => [0,8],    //1 is at key 0 and 8
  2 => [1,7],    //2 is at key 1 and 7
  3 => [2,6],    //3 is at key 2 and 6
  4 => [3,5],    //4 is at key 3 and 5
  5 => 4         //5 is at key 4
]
```

Notice how the type of the values in the resulting array above alternates. If a value only occurs once in the input array, that single position is returned in the

output array, else if there are more than one, an array of possible positions is returned.

Program 5.2 Pivot example

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

While PHP supports coercion between many of native types, there is no such thing as array coercion. The mix of arrays and scalar values in the resulting array would ultimately force a check the data type before using the values, which could have been avoided if the function consequently returned an array containing arrays of integers. Necessary data type checks strongly suggests bad design choices 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 number values, the first happens when an integer is appended to an array already containing an array or integer (figure 5.1.1), the second when an array append operation is performed on something that might be a number (figure 5.1.2).

Notice how both warnings occurs as side-effects of the real problem, namely the type of the content. In practice the sub-array always contains integers (not arrays and integers) and the append operation is always performed on a number.

```php
for ($i = 0; $i < count($source); $i++)
{
    $resultKey = $source[$i];
    $resultValue = $i;

    // The counter tracks how many times a key has been used.
    if (array_key_exists($resultKey, $counter))
    {
        // The first time around we just assign the value to the key.
        $result[$resultKey] = $resultValue;
        $counter[$resultKey] = 1;
    }
    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 append to the existing array.
        $result[$resultKey][] = $resultValue;
    }

}
```

Assigning value to array of different type

(5.1.1) Array write with different values

```php
for ($i = 0; $i < count($source); $i++)
{
    $resultKey = $source[$i];
    $resultValue = $i;

    // The counter tracks how many times a key has been used.
    if (array_key_exists($resultKey, $counter))
    {
        // The first time around we just assign the value to the key.
        $result[$resultKey] = $resultValue;
        $counter[$resultKey] = 1;
    }
    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 append to the existing array.
        $result[$resultKey][] = $resultValue;
    }
}
```

Target of write might be non-array

(5.1.2) Array append on non-array

Figure 5.1: Running the analysis on `pivot`

The real problem is however not captured by the analysis. This happens when the branches of the if-statement meets. Here the analysis preforms a join of the analysis lattice element, hence a join of the lattice element representing the abstract value of `$result` array. For the first iteration of the for-loop, the first branch of the if statement on line 11 would express the array as a list of integers, while the second branch as a list of lists containing integers. A warning on a join of these opposing representations could be implemented, but is currently not supported by the analysis.

5.3 Directory content

Found in the CodeIgniter framework, program 5.3 demonstrates mixing the map and list type arrays. This programs four library functions for traversing the file-system; `opendir` which creates a file-pointer for a given directory, `readdir` which reads the name of the next file/directory in the given directory using the file-pointer, `is_dir` which as the name suggests checks if a path is a directory, and `closedir` which *closes* the file-pointer. Since P0 doesn't support file pointers, these are modelled as numbers in the analysis.

Program 5.3 Directory content example

```
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($fp)))
8          {
9              if (($depth < 1 || $new_depth > 0)
10                 && is_dir($dir.$file)){
11                 $filedata[$file] =
12                  directory_map($dir.$file, $new_depth);


13             } else {
14                 $filedata[] = $file;
15             }
16         }
17         closedir($fp);
18         return $filedata;
19     }
20     return FALSE;
21 }
22 $result = directory_map("testDir", 2);
```

The `directory_map` function takes a directory name and a depth as input and returns the structure of the given directory recursively bound by the provided length. 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 program 5.3 provides no obvious way of using

the output of the function, and it it seem to assume that directories can't have integers as names. If a directory had a integer, i, as a name and more than i files has been seen, then adding the new folder will will override a file already added to the list. E.g. let a directory contain a file named `file1` and an empty subdirectory named `0`. If `readdir` first returns the name of the file and then the folder, the result will be `[[]]` (a list containing an empty list), if the order is reversed, the result will be `["file1", []]`.

This is a good example on how arrays in PHP serve as the go-to data-structure, when other alternatives might suit the problem better. In this case, representing the files and directories as objects would solve the problem.

The analysis raises three warnings on two code-locations. Just as the previous case, these warnings seem to have been raised as side-effects of the real problem, which is caused by the `$filedata` array being represented as a map at line 12, a list at line 14, and joined at the end of the `if`-statement. This join yields the $\top$ array which is the source of the last warning in at both code-locations: *"Target of write might be non-array"*, since it causes all heap locations to be updated when writing to `$filedata`. Yielding a warning when a join yields a $\top$ array might be a better indicator on what the *real* problem is.

►new figures◄

►consider newpage◄

```php
function directory_map($source_dir, $directory_depth = 0)
{
    if ($fp = opendir($source_dir))
    {
        $filedata = [];
        $new_depth  = $directory_depth - 1;
        while (FALSE !== ($file = readdir($fp)))
        {
            //is_dir($source_dir.$file) && $file .= '/';
            if (($directory_depth < 1 || $new_depth > 0) && is_dir($source_dir.$file))
            {
                $filedata[$file] = directory_map($source_dir.$file, $new_depth);
            }
            else
            {
                $filedata[] = $file;
            }
        }
        closedir($fp);
        return $filedata;
    }

    return FALSE;
}
```

Array write may be with string index on list
Assigning value to array of different type
Target of write might be non-array

(5.2.1) Writing with string on list and type mismatch

```php
function directory_map($source_dir, $directory_depth = 0)
{
    if ($fp = opendir($source_dir))
    {
        $filedata = [];
        $new_depth  = $directory_depth - 1;
        while (FALSE !== ($file = readdir($fp)))
        {
            //is_dir($source_dir.$file) && $file .= '/';
            if (($directory_depth < 1 || $new_depth > 0) && is_dir($source_dir.$file))
            {
                $filedata[$file] = directory_map($source_dir.$file, $new_depth);
            }
            else
            {
                $filedata[] = $file;
            }
        }
        closedir
        return $
    }

    return FALSE;
}
```

Assigning value to array of different type
Target of write might be non-array

(5.2.2) Appending a string and type mismatch

Figure 5.2: Running analysis on `directory_map` ►Crop figures◄

5.4 Date validation

Program 5.4 was found on the e-commerce platform Magento, where the original functions were located in two different files. The `isValidDate` function takes three arguments expressing a date and returns a map being either empty or containing an error message at the `"invalidDate"` key. The `isMinimumDay` similarly checks the validity of a date using the previous function, Here however, the output is a list which leads to a merge of a list and a map at line 15.

Program 5.4 Date validation example

```
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);
```

The program uses two library functions; `checkdate` and `array_merge`, modelled as a function returning boolean $\top$ and as described in section 4.8.1, respectively.

Running the analysis on the program yields a single error, illustrated as figure 5.3. As opposed to the previous cases, this error is reflecting the main problem, namely the merge of a list and a map. Whether the issue is caused by the lack of specification of the `isValidDate` function or is as designed, is not known, but it could be resolved by converting the map to a list e.g. using the `array_values` library function, or replacing append with write operations.

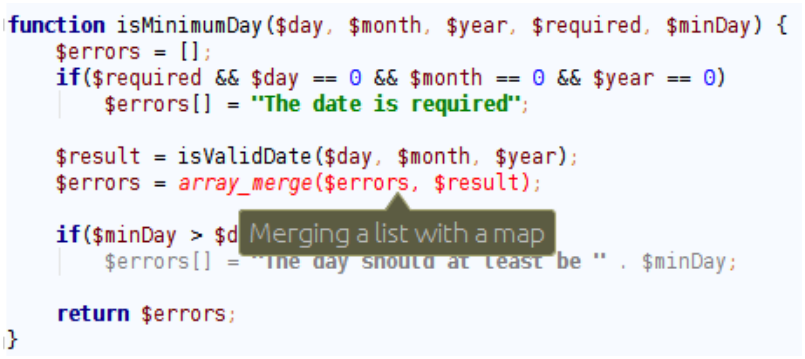


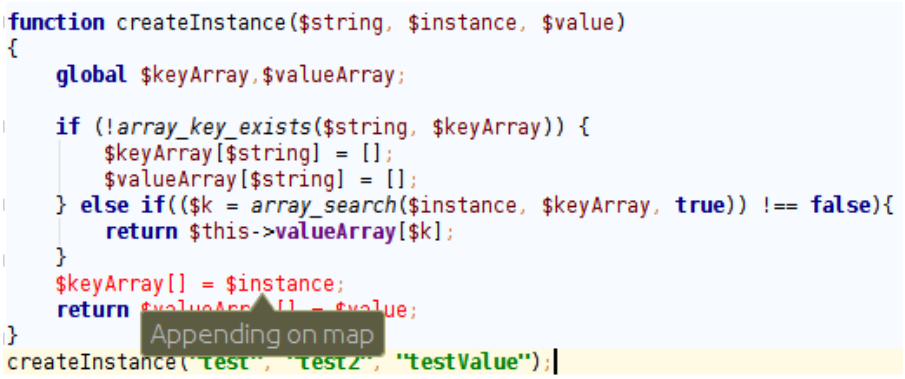
Figure 5.3: Running the analysis on date validation example ►new figure◄

5.5 Caching instances

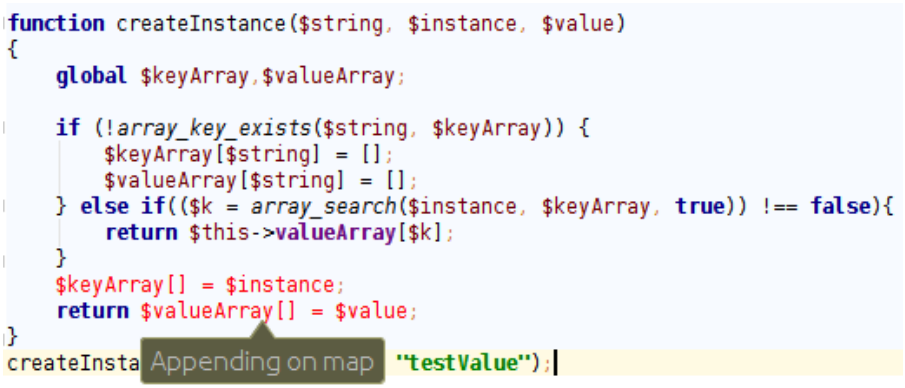
This case demonstrates the opposite of the case in section 5.3. In lines 8 and 9 the arrays associated with `$keyArray` and `$valueArray` are used in a map context by writing to a string index. Then in lines 13 and 14 the array append operation is used with both arrays. When detecting the latter the 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 fixed right away.

Program 5.5 Caching instances example

```
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,
       true)) !== false){
11         return $this->valueArray[$k];
12     }
13     $keyArray[] = $instance;
14     return $valueArray[] = $value;
15 }
16 createInstance("test", "test2", "testValue");
```



(5.4.1) Appending on map: `$keyArray`



(5.4.2) Appending on map: `$valueArray`

Figure 5.4: Running analysis on `createInstance`

Figure 5.5: Using the pop function on a map array

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 5.6 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.

Program 5.6 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 sub-captions◄

5.7 Conclusion

►Theoretical results◄

►Add an array just below the top array?◄

►Side-effect warnings◄

Since this thesis doesn't focus on efficiency wrt. memory consumption or run-time, no such information is associated with this case study. It should however be noted that execution was reasonable fast, performed in seconds rather than minutes, this in spite of the current analysis being performed after running the static analysis of the IDE. The observed memory consumption was rather high which possibly follows from the implemented caching strategy of the lattice elements.

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 alogrithm to find possible file matches and afterwards refining the results with a context sensitive program-centered alogrithm.

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.

In [**?**] variable variables are supported by treating the global scope of PHP as an array with keys corresponding to the names of variables. Accessing variable variables is then a simple matter of how array access is handlded in general.

6.2 Static Analysis

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

In this theses, a corpus consisting of some of the most popular PHP frameworks has been run and subsequently analysed, revealing that PHP arrays can in fact be categorized into two types; lists and maps, both acyclic. Furthermore it was shown that operations normally associated with lists, e.g. `array_pop`, `array_push`, `array_shift`, etc., are used almost exclusively on list-arrays. Running these programs relies on the test-suite defined by the designers of the respective frameworks, which reduces the task of analysing the corpus to adding a logging feature to the interpreter and inspecting the output. While this strategy relies on quality of the given test-suite, the size of the log-files generated (41 MB to 13 GB) and the popularity of the programs, assure us that the coverage achieved, is better than anything we could have designed.

The results of the dynamic analysis motivated the design of a classical inter-procedural data-flow analysis using the monotone framework. In this respect the main contributions are the definition of a subset language of PHP, a method of generating control-flow graphs, a context sensitive lattice expressing abstract states, and a set of argumentatively sound transfer functions. This analysis has been integrated into one of the most popular IDE for PHP programming, utilizing the AST generated by the environment and allowing instantaneous annotation of code.

While the analysis is neither proven sound, optimized for fast execution, nor executed on large programs, it has successfully annotated suspicious usage of arrays on code, rewritten to P0, discovered in the corpus in a reasonable time. This analysis serves as a steppingstone to the construction of a full-fledged static analysis of the obscure language that is PHP, and by building this analysis on basic static analysis principles, other techniques for achieving e.g. dynamic dispatch should be relatively easy to apply.

Chapter 8

Future Work

It evident, from the evaluation of the analysis developed in this thesis, that several parts of the analysis can be improved upon and further developed. The evaluation consists of small and simple case studies adapted to the supported language, P0. Even these small cases strain the resource usage and the P0 subset of PHP lack many commonly used features. In this chapter the limitations and possible improvements and further development of the analysis are discussed. The chapter is split into three sections. First discussing what is needed to extend the support from P0 to the full PHP language followed by interesting precision improvements. Precision improvements are interesting if they expand the way the analysis can produce feedback for the user, otherwise they are traded for unnecessary loss of performance. Performance, being the topic of the last section of this chapter, need improvements in order to be able to handle the size of real PHP applications at a reasonable resource usage.

8.1 Supporting the full PHP language

The static analysis developed in this thesis supports the subset of PHP, P0, as defined in chapter 4. Section 4.1 defines the restricted syntax of the P0, disallowing most dynamic features of PHP, and section 4.6 defines how abstract operators are restricted in P0.

8.1.1 Operators

Since variables are not restricted to a single type throughout a PHP program and there are no static types a lot of cross-type operations are allowed. P0 removes support for some of these cross-type operations because they are not defined in an intuitive way. To take an example the unary increment and decrement operators can be applied to strings. Listing 8.1 demonstrates how different strings are incremented. If the last character is a letter from a-y (always keeping the current case), that letter will be turned into the following letter in the alphabetical order. If the last letter is a z it will turn into an a as well as recursively applying the increment operation to the previous letter as well. If no previous letter exists an a will be prepended to the string. Any

characters that is not a letter from a-z will stop the increment operation and stay as is. While this definition seems somewhat sensible intuitively the decrement operator should then do the opposite of increment, however that is not the case. Applying the decrement operators to a string never changes the initial string.

Program 8.1 Increment and decrement operators used with strings

```
1 $a = "a";
2 $b = "Bob";
3 $c = "Z";
4 $d = "Hello World!";
5 $a++; // Result: "b"
6 $b++; // Result: "Boc"
7 $c++; // Result: "AA"
8 $d++; // Result: "Hello World"
9 // Decrementing does nothing
```

To be able to support all these operations in a sound way the official PHP Interpreter has to be studied to create an overview of how each cross-type operation is implemented and thus defined. The official PHP Interpreter is the only definition of the PHP language yet, even though work on an official definition is in progress. Until the official definition of the language is done inspecting the interpreter source code is the only way to get a full overview of how operators are implemented and use that overview to create a full set of abstract operators from value lattice element to value lattice element.

8.1.2 Dynamic features

The language subset P0 does not support variable variables. One way of supporting variable variables is to change the definition of the local and global scope lattices. Instead of mapping from variable names they could be maps from strings to values allowing an abstract operation for look-up in the global and local variable scopes. Supporting variable variables have the possibility of impacting precision in cases containing many writes to variable variables with names that cannot be reasoned about statically. Any write to a $\top$-element string have to be joined when reading from all variables which lowers the precision of all variable reads.

►write about dynamic functions◄

►anonymous functions, global and local scope is still enough to express those because of the use keyword◄

►write about includes◄

8.1.3 Objects

This thesis have completely ignored a large part of PHP, namely the object model. To be able to handle real PHP application objects must be supported since almost every PHP application is at least partly object oriented. Since objects are dynamic as the rest of PHP, properties can be added and removed at execution time. To keep soundness of the analysis object properties can

be modelled as a map array. This approach suffer from the same weakness as variable variables, that writes to $\top$-element property names lessens the precision of all other properties. By expanding the dynamical analysis from chapter 3 to log object property creations and removals a hypothesis about these operations being rare can be tested. If such a hypothesis turns out to be true an unsound disregard of property creations and removals can be applied in the analysis. Which will in turn heighten the precision of the analysis.

8.1.4 Other unsupported features

►write something about the rest of the unsupported features: resources, more library functions, what else?◄

8.2 Precision

The abstraction of a static analysis based on the monotone framework are done in two places. Firstly the lattice used to represent the state of the analysis and secondly the transfer functions or abstract operations used to transition between different lattice elements to represent changes of the program state. These abstractions are where precision is traded for other properties of the analysis like performance and ease of implementation. The next section will discuss possible improvements of the lattice abstraction whereas the rest of discusses improvements for the transfer functions.

8.2.1 Lattice improvements

Each element of the lattice represents either a certain value or a predicate restricting possible values. The analysis developed in this thesis is path-insensitive except for statically computable boolean values in conditionals where the unreachable branch will be pruned. As such the analysis lattice represents only a single predicate, i.e. whether strings, numbers, and array indices are positive integers or not. Otherwise exact values are kept where possible.

Currently the Integer-lattice, which is part of the Index-lattice, has no predicates at all. By introducing the negative and not negative (including 0) predicates differentiation between maps and lists would be more precise, since writes with non-negative integer keys would produce lists and otherwise a map will be produced.

8.2.2 Arrays

The current implementation of the array read transfer function uses a naive handling of reading from $\top$-element arrays that simply results in all existing heap locations. A possible way to improve on this naive solution would be to result in all heap locations that are part of an array. To get an indication of whether this is a useful precision improvement existing code which results in $\top$-element arrays should be studied to find examples where the more precise

implementation leads the analysis to discover suspicious array uses otherwise not detected.

8.2.3 References

Because of PHP keeping references when copying arrays the analysis can not soundly perform strong updates however references inside arrays may be one of the less used features of PHP. A hypothesis about the use of references in arrays can be tested by expanding the dynamic analysis in chapter 3. If references in arrays turn out to be rarely used strong updates can be reintroduced. The unsoundness introduced by strong updates would then be negligible compared to the precision gained.

8.3 Performance

Performance concerns can be split into two parts, namely theoretical performance and practical performance. First-mentioned is the performance of the specified analysis whereas the latter is optimizations in the implementation while still conforming to the specification of the analysis.

►write something about resource usage for the case studies (CPU, memory)◄

8.3.1 Analysis performance

►does this split even make sense? What can be improved in the analysis specification performance-wise?◄

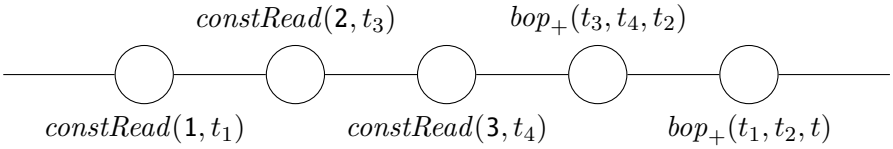
8.3.2 Implementation performance

No performance optimizations have been made on the implementation of the analysis since performance have not been the primary concern.

The analysis lattice is the primary data-structure used and thus a major factor in performance regarding to memory usage. To obtain context sensitivity the State-lattice is stored multiple times in the analysis lattice and because of this minimizing the size of the State-lattice results in multiple times the performance gain.

The two lattice-parts Temps and HeapTemps are used to pass information between control-flow nodes and are specified as partial maps from temporary variable names and temporary heap variable names respectively. Keeping only specified entries in the map and returning a set “empty” value for unspecified entries keeps the memory usage reasonable. The current implementation never removes entries from the map which will keep growing in size with programs of larger size. Due to the nature of how these maps are used each entry can be seen as having a provider and consumer control-flow node. The provider node sets the entry which is then later used by the consumer node. To keep the maps small in size even with large programs the consumer nodes can remove the entries they consume. Graph 8.1 shows a simple control-flow graph to

demonstrate the effect of letting the consumer node remove consumed entries. In listing 8.2 the first line shows the entries which would be set in the Temps lattice-element associated with the last node without any optimizations. The second line shows the set entries with the optimization.



Graph 8.1: Example graph $[\![\texttt{1+(2+3)}]\!](t)$

Program 8.2 Set entries in the Temps part of the lattice-element associated with the last $\texttt{bop}_+$-node before and after optimizations

```
1 temps → [t1, t2, t3, t4, t] // without optimization
2 temps → [t] // with optimization
```

8.4 Summary

In this chapter we have proposed different ways to continue the work done in this thesis. To be able to use the analysis developed in chapter 4 in any real context the full PHP language have to be supported and performance have to be improved to be able to analyse larger programs. For the former we have proposed ways to handle dynamic features, the remaining cross-type operations, objects, resources and library functions. For the latter we have proposed a method to keep the lattice-parts concerned with temporary variables small in size regardless of the size of the program. We have further more proposed possible precision improvements at the cost of soundness if hypotheses about the use of references and creations/removals of object properties can be confirmed.

Appendix A

Basic Definitions

Definition 10. 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 11. 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 12. The sum of two lattices, $L_1 = (E_1, \sqsubseteq_1)$ and $L_2 = (E_2, \sqsubseteq_2)$ where $\{\top, \bot\} \subseteq E_1 \cap E_2$, is defined as

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

Where

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

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

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

►**Lemma:** L_{Sum} **is a lattice.** ◄

Definition 13. The product of two lattices, $L_1 = (E_1, \sqsubseteq_1)$ and $L_2 = (E_2, \sqsubseteq_2)$, are defined as

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

Where

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

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

$$e_1 \sqsubseteq e_2 \Leftrightarrow x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2 \tag{A.6}$$

►**Lemma:** L_{Prod} **is a lattice.** ◄

Definition 14. Given a set, $A = \{a_1, \cdots, a_n\}$, and a lattice $L = \{E, \sqsubseteq\}$, a map lattice is defined as

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

Where

$$E_{Map} = \{([a_1 \mapsto x_1, \cdots, a_n \mapsto x_n] | x_i \in E\} \tag{A.8}$$

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

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

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

►**Lemma:** L_{Map} **is a lattice.** ◄

Definition 15. Given a set A, the powerset is defined as

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

Where

$$E_P = \{S | S \subseteq E\} \tag{A.11}$$

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

$$e_1 \sqsubseteq_P e_2 \Leftrightarrow e_1 \subseteq e_2 \tag{A.12}$$

►**Lemma:** $P(A)$ **is a lattice.** ◄

Appendix B

Abstract Operators

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

$$x, y \in \mathbb{Z} \land x, y > 0$$
$$a, b \in \mathbb{R} \land (a, b < 0 \lor a, b \notin \mathbb{Z})$$

Furthermore the shortshand f is used for the boolean `false` value.

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

Table B.1: Abstract addition

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

Table B.2: Abstract subtraction

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

Table B.3: Abstract multiplication

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

Table B.4: Abstract division

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

Table B.5: Abstract modulus

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

Table B.6: Abstract exponentiation

==	⊥	t	f	⊤
⊥	⊥	⊥	⊥	⊥
t	⊥	t	f	⊤
f	⊥	f	t	⊤
⊤	⊥	⊤	⊤	⊤

(a) Equality

!=	⊥	t	f	⊤
⊥	⊥	⊥	⊥	⊥
t	⊥	f	t	⊤
f	⊥	t	f	⊤
⊤	⊥	⊤	⊤	⊤

(b) Not Equality

<	⊥	t	f	⊤
⊥	⊥	⊥	⊥	⊥
t	⊥	f	f	f
f	⊥	t	f	⊤
⊤	⊥	⊤	f	⊤

(c) Less Than

<=	⊥	t	f	⊤
⊥	⊥	⊥	⊥	⊥
t	⊥	t	f	⊤
f	⊥	t	t	t
⊤	⊥	t	⊤	⊤

(d) Less Than/Equal

>	⊥	t	f	⊤
⊥	⊥	⊥	⊥	⊥
t	⊥	f	t	⊤
f	⊥	f	f	f
⊤	⊥	f	⊤	⊤

(e) Greater Than

<=	⊥	t	f	⊤
⊥	⊥	⊥	⊥	⊥
t	⊥	t	t	t
f	⊥	f	t	⊤
⊤	⊥	⊤	t	⊤

(f) Greater Than/Equal

Table B.7: Abstract comparison operators on booleans

==	⊥	y	uIntString	s	notUIntString	⊤
⊥	⊥	⊥	⊥	⊥	⊥	⊥
x	⊥	x==y	⊤	f	f	⊤
uIntString	⊥	⊤	⊤	f	f	⊤
r	⊥	f	f	r==s	⊤	⊤
notUIntString	⊥	f	f	⊤	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤	⊤

Table B.8: Abstract Equal for Strings

!=	⊥	y	uIntString	s	notUIntString	⊤
⊥	⊥	⊥	⊥	⊥	⊥	⊥
x	⊥	x!=y	⊤	t	t	⊤
uIntString	⊥	⊤	⊤	t	t	⊤
r	⊥	t	t	r!=s	⊤	⊤
notUIntString	⊥	t	t	⊤	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤	⊤

Table B.9: Abstract Not Equal for Strings

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

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

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

Table B.11: Abstract equal on numbers

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

Table B.12: Abstract not equal on numbers

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

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

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

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

&&	$\bot$	t	f	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
t	$\bot$	t	f	$\top$
f	$\bot$	f	f	f
$\top$	$\bot$	$\top$	f	$\top$

(a) `AND`

||	$\bot$	t	f	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
t	$\bot$	t	t	t
f	$\bot$	t	f	$\top$
$\top$	$\bot$	t	$\top$	$\top$

(b) `OR`

XOR	$\bot$	t	f	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
t	$\bot$	f	t	$\top$
f	$\bot$	t	f	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$

(c) `XOR`

Table B.15: Abstract logical binary boolean operators

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

(a) Abstract pre- and post-increment

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

(b) Abstract pre- and post-decrement

-	$\bot$	n	uInt	notUInt	$\top$
	$\bot$	$-n$	notUInt	$\top$	$\top$

(c) Unary minus

!	$\bot$	true	false	$\top$
	$\bot$	false	true	$\top$

(d) Boolean negate

Table B.16: Abstract unary operators