# TAPAs: Type Analysis for PHP Arrays


Christian Budde Christensen, 20103616

Randi Katrine Hillerøe, 20103073


Master's Thesis, Computer Science

May 2015

Advisor: Anders Møller


AARHUS UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

# Abstract


**▶in English. . . ◀**

# Resumé

▶in Danish. . .◀

# Acknowledgements

►. . . ◄

*. . .*
*Aarhus, May 28, 2015.*

# Contents

# Chapter 1

# Introduction

PHP is one of the most popular languages for server-side web-development. It is used by major websites, such as Wikipedia and Facebook, are powering over 80% of the web[1], 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, and hosting cheap.

As other dynamically typed languages, static type reasoning is non-trivial and is only complicated by the all-purpose array datastructure. PHP supports associative arrays with integer and/or string-typed indices (referred to as keys) and anything, specially arrays, as possible values. Combined with the dynamic type system, extensive coercion, and optional error-reporting bug-finding is 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 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 defined. The program is thus valid PHP, but suspicious to say the least.

[1] `http://w3techs.com/`

In the following real-life example, from the Part framework, the `$keyArray` and `$valueArray` arrays are first used as a map in lines 4 and 5 while later, at 9 and 10 with the *array append* operation, being used as a list. 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 cases as suspicious.

**Program 1.1** Valid, but ugly, program

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

**Program 1.2** Other type

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

## 1.1 Problem Statement

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

## 1.2 Motivation

Our hypothesis is 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 to understand why arrays are difficult to apply existing methods to. In chapter 4.3

a dynamic analysis is conducted to identify use-patterns and test the hypothesis about use-patterns of arrays. Chapter 4 use patterns and knowledge gained in the previous chapter to define and implement a static analysis of PHP arrays which detects suspicious use of arrays. The static analysis implementation is evaluated in chapter 5 and relevant related work is described in chapter 6. The last two chapters 7 and 8 concludes our thesis and describes possible future work identified.

# Chapter 2

# Background

PHP is a recursive acronym meaning PHP: Hypertext Processor. As one of the first dedicated web development server-side languages it has built a huge user base over the past 20 years. A survey[?] shows that 82% of websites use PHP. Whereas PHP is used for both high traffic websites like Facebook and Wikipedia and low traffic websites, the nearest contestants tends to be used for high traffic sites. The simplicity and availability of the language allows for creating small websites with PHP even with no prior programming experience. The wide range of easy-to-use content management systems, forum application and blogging platforms also enables the use of PHP for a variety of users. All these facts makes it impossible to expect well-structured and organized use of the language and all of its possibilities.

## 2.1 Arrays in PHP

The language construct which is called `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.

## 2.2 Scoping

The scoping of PHP is rather simple; the only existing local scope is inside of function bodies. All blocks inherit their parent scope and thus the developer need not worry about at which block level a variable is first declared or used. The developer should only be aware when declaring functions. PHP offers the `global` keyword to allow access to variables defined in the global scope inside of function bodies. A few variables are always accessible inside the local function scope these are called `superglobals` and consists of `$_POST` and `$_GET` for fetching user input from HTTP POST and HTTP GET requests respectively, `$_SERVER` for accessing server environment information and a few others. Listing 2.1 shows an example of using `superglobals`. **▶Other super-globals◀**

**Program 2.1** Global variables used in function scope

```
session_start();
$username = "";

function setUser($ID, $name) {
    global $username;
    $_SESSION["ID"] = $ID;
    $username = $name;
}

setUser(1, "Admin");

echo "Hello $username"; // Result: Hello Admin
```

A superglobal to be aware of is the `$GLOBALS` array which is an array containing all global variables with the variable names as keys. This array can be used to access global variables in function scope without using the `global` keyword.

## 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 variables 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 listing 2.2.

**Program 2.2** Changing a reference by reference is not possible

```
$hello = "world";
$hello2 = "stupid";

function change(&$input) {
    $input = &$GLOBALS["hello2"];
    $input = "awesome";
}

change($hello);
// Result: $hello2 = "awesome" and $hello remains
    unchanged
```

Knowing how PHP splits handle names and content as two different concepts assignments 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 strategy. In listing 2.3 after line 3 both `$a`, `$b`, and `$c` points to the same array. After evaluating line 4 the array is copied, updated and `$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 listing 2.4

**Program 2.4** Aliasing

```
function byvalFunc($input) {
    $input["hello"] = "PHP";
}

function byrefFunc(&$input) {
    $input["hello"] = "PHP";
}

$greet = ["hello" => "world"];
byvalFunc($greet);
echo $greet["hello"]; // Result: world
byrefFunc($greet);
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 listing 2.5.

**►expand PHP section with txt◄ ►Write about PHP pograms (e.g. phpunit, composer etc◄ ►Write about vagrant and test setup ◄ ►write about lattice theory◄ ►write about monotone framework◄**

**Program 2.5** Keeping references in arrays

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

# Chapter 3

# Dynamic Analysis

The possibilities of 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 a few more restrictive data types contained within the way PHP arrays are used. The results of the analysis are used to choose 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 as figure 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, are, 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

```php
$numbers = [1,2,3];
$numbers[] = 4; // $numbers = [1,2,3,4]
$first = array_shift($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 all integer keys 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 figure 5.1 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

```php
$text_to_int =
    [
        'one' => 1,
        'two' => 2,
        'three' =>  3
    ];
echo $text_to_int[$input];
$keys = array_keys($text_to_int);
    // $keys = ["one", "two", "three"]
$values = array_values($text_to_int);
    // $values = [1,2,3]
```

If given an map-array, the values or keys can be fetched using the functions `array_values` or `array_keys` respectively. These functions returns an array of 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, this include 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 type. Then $a$ can be considered an array of type object.

## 3.1 Hypothesis

Our hypothesis is 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 that append, push, pop, shift and unshift operations are only used on arrays of type list.

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 progrms 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 database[**?**].
- *MediaWiki*: The framework for creating wiki sites[**?**].
- *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 library functions (`array_push()`, `array_pop()`, `array_search()`, `count()`, etc.), array reads, array writes, assignments, 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 are starting with a *line type*, which identifies the type of the entry. The possible line types are described in the list below.

- *array function*: Every call to an library array function[4], such as `array_push()` or `count()` are logged with the function name as line type. The array functions does 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, 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, on the form `$x[$key] = $y`, are logged with the array being read from, `$x`, the key, and the `$key`.
- `array_append`: When elements are added to the array using the append method, `$x[] = $y`, this is logged with the array being read from, `$x`.
- `assign_*`: Every assignment is logged as either `assign_const`, `assign_tmp`, `assign_var` or `assign_ref` depending wheteher 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, in a function 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 in the array as well as whether it contains any self-references. Lists contains sequential integer keys starting from 0, Sparse List is any other arrays with only

[4] `http://php.net/manual/en/ref.array.php`

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

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

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

(3.1.2) File: `a.php`

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

integer keys and Map is all arrays with at least one key of type string. Any array with a self-reference is Cyclic and all other arrays are Acyclic.

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

Each entry, but the `hash_init` entries, contains a line number and a file path to where the action occurred.

### 3.2.2 Identification of arrays

In order to analyze how arrays are used throughout a log file, some method for identifying which arrays are mentioned on a given line is needed. E.g. in the output 3.1.1 all lines are concerning 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 the a new array is initialized at some address. If such a line occurs between two usages of an address they can not necessarily be considered representing the same array.

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

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

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

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

```
$ 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.

```
rb.enable_debug=1
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 Discussion

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 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 does rely on the type of the values read or written to the arrays. This implies that there is no reasoning about entries or arrays never read or written. This might produce some false positives.

- Callables can be written as anonymous functions, strings, or arrays, containing an instance and a string. This analysis will fail to classify arrays containing callables, written differently, correctly, thus introducing false negatives. E.g in example 3.3; `$callable1`, `$callable3` and `$callable4` are all valid callables, however `$callable2` is not, since `$a->f2()` is a private function.

[5] `http://vagrantup.com/`

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

**Program 3.3** Callables in PHP

```
class A{

    public function f1(){
        ...
    }

    private function f2(){
        ...
    }

}

function f(){
    ...
}

$a = new A();

$callable1 = [$a, "f1"];
$callable2 = [$a, "f2"];
$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 3.3.2 line 5. This follows from the value first being assigned to the sub-array which then is assigned to the super-array. This leads to false negatives and could be solved if the interpreter kept character location information in addition line number and file.

## 3.3 Results

Table 3.1 shows that across all frameworks in the corpus less than 1% of the arrays are detected as being cyclic. Cyclic arrays are created using the the PHP reference operator, &, which must be used explicitly. Due to the explicit reference operator cyclic arrays does not occur as an unintentional side effect of something else►**something else?**◄. The largest amount of cyclic arrays detected in the corpus is PhpMyAdmin with a total of 10 cyclic arrays out of 3,373 identified arrays. By assuming that arrays are acyclic the static analysis would not have to take recursive types into consideration. Since the results shows that almost every framework contains some cyclic arrays the static analysis must handle recursive types in some way. The small amount of cyclic arrays however does imply that an imprecise handling of recursive types should not

```
hash_init        0x2A
array_init       3          b.php     0       1       0       0x2A
hash_init        0x2B
array_write      5          b.php     0       1       0       0x2B   long
          1     NULL
hash_init        0x2C
array_write      5          b.php     0       1       0       0x2C   long
          2     NULL
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

impact the overall precision by much. **▶Rewrite this with new findings. See comment.◀**

| Framework | # Arrays | # Cyclic arrays | # NG Cyclic arrays |
|---|---|---|---|
| Code igniter | 331 | 0 | 0 |
| Joomla | 1969 | 2 | 0 |
| Magento2 | 6942 | 0 | 0 |
| Mediawiki | 27368 | 1 | 0 |
| Part | 378 | 0 | 0 |
| phpBB | 2529 | 1 | 0 |
| PhpMyAdmin | 3373 | 10 | 0 |
| Symfony | 3707 | 6 | 6 |
| Wordpress | 3054 | 1 | 0 |
| Zend Framework 2 | 4381 | 3 | 2 |

Table 3.1: Amount of cyclic arrays detected in the corpus

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

The Object group marked with List is by the definition categorized as objects, but they might fit better into the List category, as lists of a top level type. **▶elaborate◀**

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

```php
$a = [];
$a[0] = 42; // array write
```

| | 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

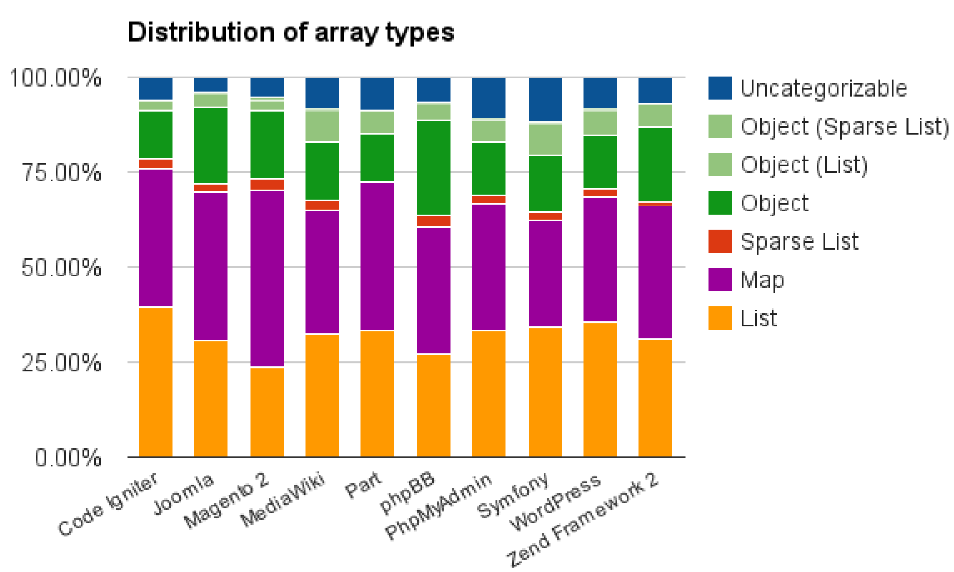



Figure 3.4: Distribution of different types of arrays

```
3 $a[] = 1337; // array append
```

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

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

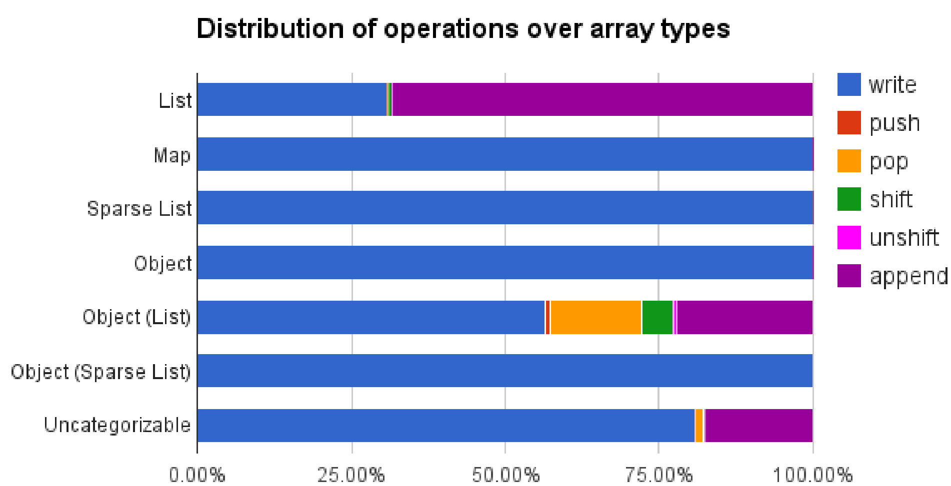



Figure 3.5: Distribution of array-changing operations over array types

## 3.4 Conclusion

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

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

A significant amount of arrays turned out to be objects with list-keys by the initial definition. These objects indicates that the object type is not providing significant information in itself, and shows the possibility that the definitions of maps and lists consume the object type; i.e. letting maps and lists allow values of different type.

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

# Chapter 4

# Data Flow Analysis

**►Outline the chapter. Explain why the sections are relevant◄**

## 4.1 PHP Language subset

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

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

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

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

**►elaborate◄**

**►Write about support of alias◄**

⟨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⟩ ⊕ ⟨expr⟩
| ∘ ⟨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

## 4.2 Control flow graph

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

**Definition 6.** A control-flow-graph $G = (V, E, s, t)$ where $V$ is a set of nodes, $E$ is a set of node pairs representing an edge between two nodes, $s \in V$ is a start node and $t \in V$ is an exit node. When illustrated as a graph, the entry node is marked with an ingoing edge with no origin and the exit node is marked with an outgoing edge with no target. E.g. in example graph 4.1, $s = const_r(1, t_1)$ and $t = bop_+(t_1, t_4, t)$.

There are seventeen different nodes, all introduced below

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

$bop(t_l, t_r, t_{tar})$, $sop(t_l, t_r, t_{tar})$, $uop(t, t_{tar})$, $inc(h, t_{tar})$: These nodes indicates binary, short-circuit-binary, uanry, and increment/decrement operations, respectively. The operations of the first three are all performed on temporary storage, while the forth is performed on the heap. The last argument, $t_{tar}$, indicates where the result of performing the operation should be stored. E.g. `1+2` is a binary operation ($bop$) returning `3`, this value should be stored in temporary storage at $t_{tar}$.

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

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

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

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

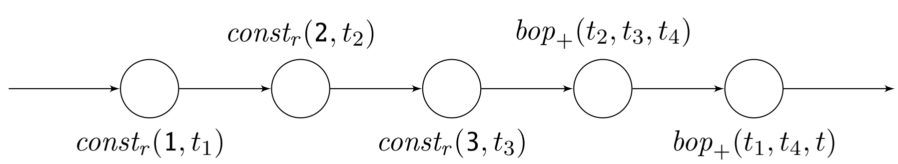
$array_a(t_{val}, t_{ar})$, $array_a(h_{var}, c_{val}, t_{tar})$, $array_a(h_{var}, h_{tar})$: With three different signatures, this node represents appending to an array, in temporary storage, in the heap, and in a read-context. Normally PHP will not allow appending without a value, but in this case,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

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

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

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

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




Graph 4.1: Example graph $[\![$`1+(2+3)`$]\!](t)$

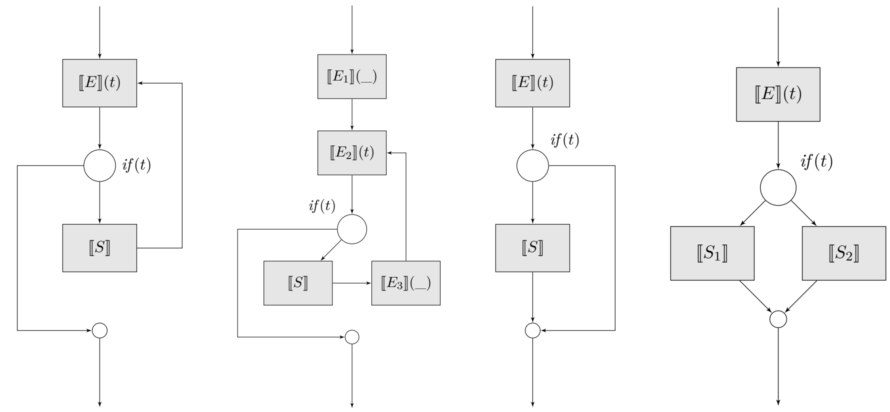
The control-flow graph created from $p$ starts with a *start* node followed by the control-flow graphs for each statement in $p$, connected edges from exit to entry nodes, in the order of appearance.

For each function definition encountered a separate graph is created, which may later be referenced. This graph starts with a *start* node and ends with an *exit* node, between these are the graph corresponding to the function body. The arguments of the exit node are the temporary variables created by the return statements.

### 4.2.1 Statements

Let $s : \langle statement \rangle$ 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.




(4.2.1) `while`-statement (4.2.2) `for`-statement (4.2.3) `if`-statement (4.2.4) `if-else`-statement

Return-statements, $s$ = **return** $T$ have three different graphs. If the return statement is empty, i.e. does not contain an expression, then this is equivalent of returning **null**, this case yields the graph in 4.3.1. If $T : \langle rexpr\rangle$. If 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), assuming the program is a valid P0 program, then it is fair to assume that $T$ can be parsed as an expression, since it cannot be an array-append operation. If it was an array append operation, then the program wouldn't be a valid PHP program and thus not a valid P0 program (see section 4.1). Assuming that the $T : \langle expr\rangle$ the graphs for a return statement should evaluate $T$ and return the result, this is the case in graph 4.3.2 with $c$ : Temps. For all graphs, the exit-node is the unique exit-node of the function.

**Program 4.1** Return-statement examples

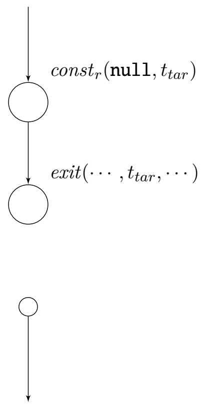
```
function normRet(){
  $a++;
  return $a;
}

```

**(4.1.1)** Normal return-statement

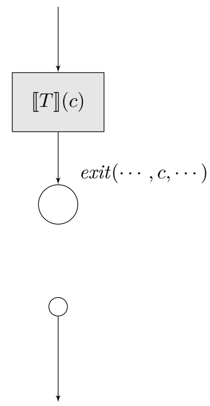
```
function &refRet(&$a){
  return $a[];
}


```

**(4.1.2)** Reference return-statement

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




(4.3.1) **`return`**-statement



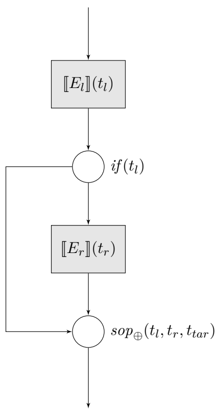

(4.3.2) **`return`**-expression-statement

### 4.2.2 Expressions

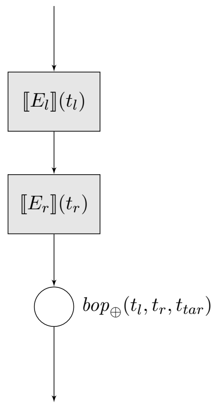
Let $e : \langle expr \rangle$ then, by ignoring the trivial parenthesized expression case, viewing the four increment/decrement operations as one, and the two array-initialization operations as one, the expression can be on nine different forms. For $e = E_l \oplus E_r$ the graph depends on the operation.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

For a variable read, $e = \texttt{\$v}$ for some variable `$v`, or a constant read, e.g. $e = \texttt{"foo"}$, the graph is a single $var_r(\texttt{\$v}, t_{tar})$ or $const_r(\texttt{"foo"}, t_{tar})$ 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 $E_{ar}$ and $c_{ar}, c_{tar}$ : Temps. If the expression is an array initialization, $e = \texttt{[}\cdots, E_1, \cdots, E_2\texttt{=>}E_3\texttt{]}$, 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.

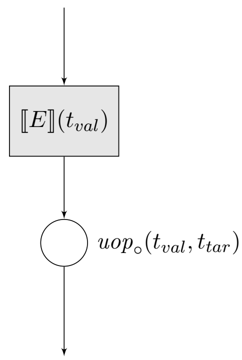
For function calls, $e = fn\texttt{(}T_1, \cdots, T_n\texttt{)}$, the result variable is a temporary variable, $c_{tar}$ : Temps, the arguments are either an expression, $T_i : \langle expr \rangle$ and $c_i$ : Temps, or an reference expression, $T_i : \langle rexpr \rangle$ and $c_i$ : HeapTemps, depending on the signature of *fn*. If the argument is pass-by-reference i.e. the



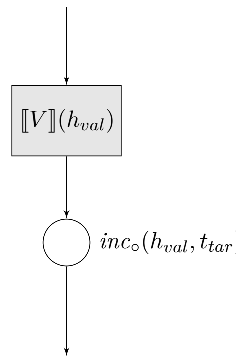

(4.4.1) Short-circuit binary operation



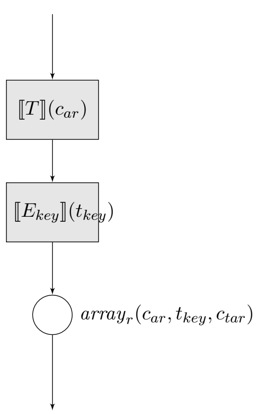

(4.4.2) Binary operation



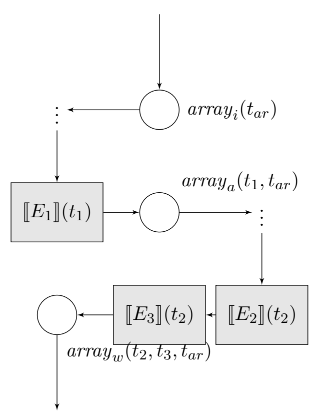

(4.4.3) Unary operation




(4.4.4) Post/pre increment/decrement opreation




(4.5.1) Array read

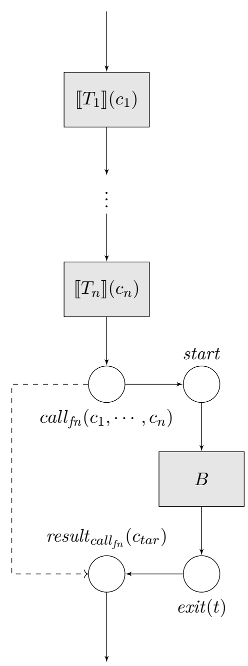



(4.5.2) Array initialize

variable, $v_i$ is denoted with a & in the signature, then the expression must be a reference expression, if not, then the argument is an expression. This follows from the program being a valid P0 program. The function graph will be as graph 4.6.1, where the start node, exit node, and function body are the unique nodes of the *fn* function i.e. they are not copied. This ensures that the graph is finite, but forces the introduction of the notion *valid successors*. Multiple, say $n$, calls to the same function will yield a graph with $n$ edges to the start node and from the exit node, thus indicating that the flow may jump to an arbitrary exit. This is naturally not the case and a successor, $w$, to a node, $v$, is thereby only valid iff. $v$ is an exit node and $w$ is the return node corresponding to the call node of the current function-call, or if $v$ is not an exit node (See def. 7). Notice the dashed line going directly from the call node to the return node. This indicates that the call node may pass information directly to the return node, typically information regarding the local context before calling *fn*.

**Definition 7.** The predicate *validSuccessor* : $(\mathcal{N} \times \Delta) \times (\mathcal{N} \times \Delta)$, is defined as

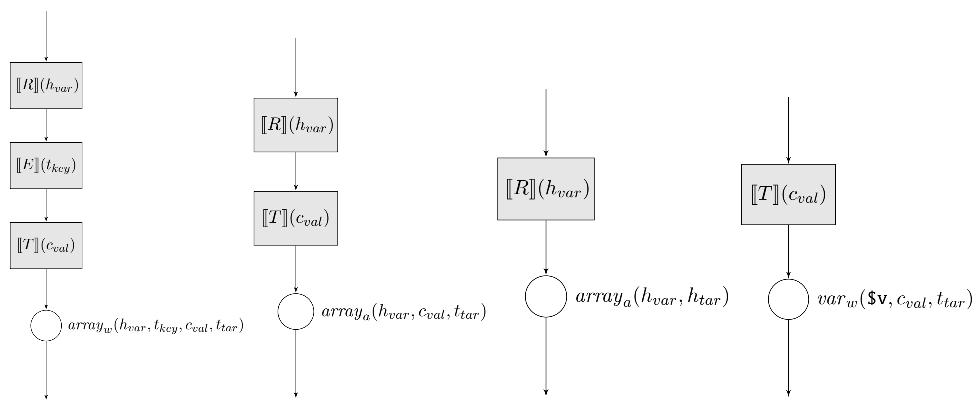
$$\begin{aligned} \mathit{validSuccessor}(n, \delta, n', \delta') \Leftrightarrow & (n = \mathit{exit}(\_) \wedge n' = \mathit{result}_c(\_) \wedge \delta = (\delta' c)) \\ & \vee n \neq \mathit{exit}(\_) \end{aligned} \tag{4.1}$$




(4.6.1) Function call

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

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




(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

```php
<?php

$a = []; //Value assignment
$a[] = 1; //Array append assignment
$a[1] = 2; //Array write assignment

$b = &$a; //Value reference assignment
$c[] = &$a;//Array append reference assignment
$d[1] = &$a; //Array write reference assignment
```

### 4.2.3 Reference and variable expressions

In order to resolve the variable being modified reference- and variable-expressions are introduced. The only difference between the two is that reference expressions may contain function references. For $r = fn(c_1, \cdots, c_n)$ the graphs corresponds to graph 4.6.1, with heap-locations as result, i.e. $c_{tar}$ : HeapTemps. For the variable read, $r =$ `$v`, the graphs is a single $var_r($`$v`$, h_{tar})$ node. When the expression is an array-read, $r = R_{ar}[E_{key}]$, the graph corresponds to that of 4.5.1 with $c_{ar}, c_{tar}$ : HeapTemps and $T$ being the graph of $R_{ar}$. Finally for the append operation, $r = R_{ar}[]$, the graphs corresponds that of graph 4.7.3.

## 4.3 Lattice

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

$$\text{AnalysisLattice} = \Delta \to \text{State} \tag{4.2}$$

This is a map from a context to an abstract state, where the context, $\Delta$ consists of a list of call-sites, represented as the call nodes in the control flow graph, with length bounded by $k > 0$.

$$\Delta = \text{CallNode}_*^{\leq k} \tag{4.3}$$

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

$$\text{State} = \text{Locals} \times \text{Globals} \times \text{Heap} \times \text{Temps} \times \text{HeapTemps} \tag{4.4}$$

As described, in section 2.2, the scoping rules of PHP are very simple and can be expressed with a global and a local scope. The global scope is necessary, because global variables can always be accessed from a function using the `global` statement. Furthermore the super-global variables resides in the global scope but can always be accessed directly. Only two scopes is enough, because any other variables has to be passed to a function as an argument. This is even the case for anonymous functions, where however the `use` keyword can be used to pass variables to a function when defining the function.

$$\text{Locals} = \text{Globals} = \text{Scope} = \text{Var} \to \mathcal{P}(\text{HLoc}) \tag{4.5}$$

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

$$\text{Heap} = \text{HLoc} \to \text{Value} \tag{4.6}$$

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

$$\text{Temps} = \text{TVar} \to \text{Value} \tag{4.7}$$

$$\text{HeapTemps} = \text{THVar} \rightarrow \mathcal{P}(\text{HLoc}) \tag{4.8}$$

The necessity for the latter lattice follows from the fact that reference assignments may be nested, which requires intermediate results shared between nodes in the control-flow graph. The sets of temporary variables, **TVar** and **THVar**, are both finite, following from the control-flow-graph being finite.

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

$$\text{HLoc} = \Delta \times \mathcal{N} \times \mathbb{N} \tag{4.9}$$

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

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

$$\text{Value} = \text{Array} \times \text{String} \times \text{Number} \times \text{Boolean} \times \text{Null} \tag{4.10}$$

Following the results of the dynamic analysis in , the array is considered either a set of locations or a map from indices to sets of types. I.e. lists or maps. The sum-lattice has been chosen as opposed to a product-lattice, following the study in chapter 4.3, which indicates that arrays seldom changes from lists to maps or vice versa. Top array elements is then a predictor for suspicious behavior. Furthermore the array lattice has an element for the empty array which can become either a list or a map.

$$\text{ArrayList} = \mathcal{P}(L) \tag{4.11}$$

$$\text{ArrayMap} = \text{Index} \rightarrow \mathcal{P}(L) \tag{4.12}$$

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

$$\text{Index} = \text{String} + \text{Integer} \tag{4.13}$$

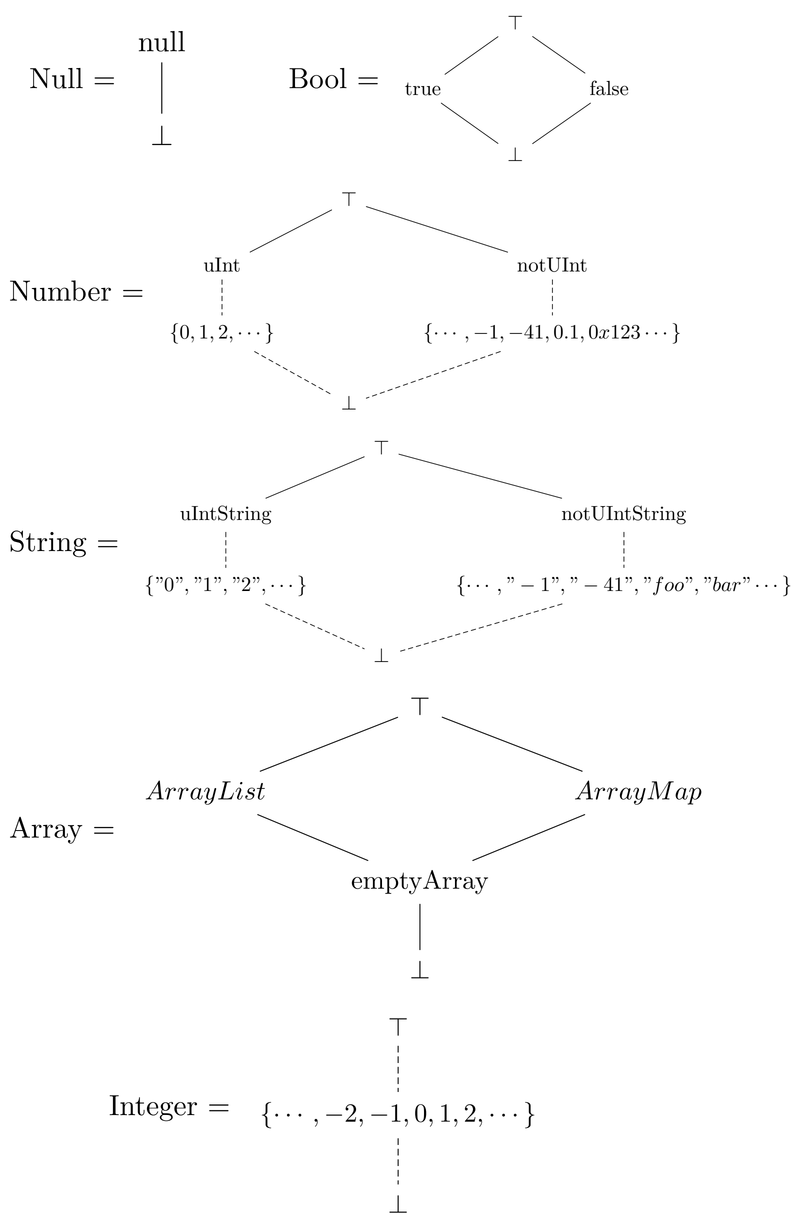



Diagram 4.1: Hasse diagrams of lattices

## 4.4 Transfer functions

**►All these overwriting of target in temps, are they any good?◄**

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

$$f_{n,\delta}(l) = l[\delta \mapsto f'_{n,\delta}(l(\delta))] \tag{4.14}$$

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

### 4.4.1 Operations

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

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

or $n = uop_{\circ}(t_{val}, t_{tar})$ then

$$f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = (s_l, s_g, s_h, s_t[t_{tar} \mapsto \circ s(t_{val})], s_{ht}) \tag{4.16}$$

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

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

$$updateLocations(L, h, v) = h[\forall l \in L . l \mapsto h(l) \sqcup v] \tag{4.17}$$

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

$$
\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \texttt{match } \circ \\
& \quad \texttt{with } \text{PreIncrement:} \\
& \quad \texttt{with } \text{PreDecrement:} \\
& \quad\quad \texttt{let} \\
& \quad\quad\quad s'_h = \mathit{updateLocations}( \\
& \quad\quad\quad\quad s_{ht}(h_{val}), s_h, l \to \circ s_h(l)) \\
& \quad\quad \texttt{in} \\
& \quad\quad (s_l, s_g, s'_h, s_t[t_{tar} \mapsto s'_h(s_{ht}(h_{val}))], s_{ht}) \\
& \quad \texttt{with } \text{PostIncrement:} \\
& \quad \texttt{with } \text{PostDecrement:} \\
& \quad\quad (s_l, s_g, \\
& \quad\quad\quad \mathit{updateLocations}(s_{ht}(h_{val}), s_h, l \to \circ s_h(l)), \\
& \quad\quad\quad s_t[t_{tar} \mapsto s_h(s_{ht}(h_{val}))], s_{ht})
\end{aligned} \tag{4.18}
$$

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

### 4.4.2 Variables

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

$$
\begin{aligned}
\mathit{writeVar}_{n,\delta}(v, s, h, v_{val}) = & \texttt{if } s(v) = \emptyset \texttt{ then} \\
& \quad (s[v \mapsto \{\text{HLoc}(n, \delta, 0)\}], h[\text{HLoc}(n, \delta, 0) \mapsto v]) \\
& \texttt{else} \\
& \quad (s, \mathit{updateLocations}(s(v), h, v_{val}))
\end{aligned} \tag{4.19}
$$

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

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

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

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

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = &\texttt{if } \delta = \Lambda \vee \mathit{isSuperGlobal}(v) \texttt{ then} \\
&\quad (s_l, s_g[v \mapsto s_{ht}(h_{val})], s_h, s_t[t_{tar} \mapsto s_h(s_{ht}(h_{val}))], s_{ht}) \\
&\texttt{else} \\
&\quad (s_l[v \mapsto s_{ht}(h_{val})], s_g, s_h, s_t[t_{tar} \mapsto s_h(s_{ht}(h_{val}))], s_{ht})
\end{aligned} \tag{4.21}$$

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

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

$$\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = &\texttt{let } s = \\
&\quad \texttt{if } \delta = \Lambda \vee \mathit{isSuperGlobal}(v) \texttt{ then } s_g \texttt{ else } s_l \\
&\texttt{in} \\
&\texttt{let } v = \\
&\quad \texttt{if } s(v) = \emptyset \texttt{ then } \mathrm{Value}(\mathrm{Null}(\top)) \texttt{ else } s_h(s(v)) \\
&\texttt{in} \\
&(s_l, s_g, s_h, s_t[t_{tar} \mapsto v], s_{ht})
\end{aligned} \tag{4.22}$$

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

$$\mathit{initializeVariable}_{n,\delta}(v, s) = \texttt{if } s(v) = \emptyset \texttt{ then } s[v \mapsto \{\mathrm{HLoc}(n, \delta, 0)\}] \texttt{ else } s \tag{4.23}$$

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

$$
\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \texttt{if } \delta = \Lambda \vee \mathit{isSuperGlobal}(v) \texttt{ then} \\
& \quad \texttt{let } s'_g = \mathit{initializeVariable}_{n,\delta}(v, s_g) \texttt{in} \\
& \quad (s_l, s'_g, s_h, s_t, s_{ht}[h_{tar} \mapsto s'_g(v)] \\
& \texttt{else} \\
& \quad \texttt{let } s'_l = \mathit{initializeVariable}_{n,\delta}(v, s_l) \texttt{in} \\
& \quad (s'_l, s_g, s_h, s_t, s_{ht}[h_{tar} \mapsto s'_l(v)]
\end{aligned} \tag{4.24}
$$

### 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 $n = array_i(t_{tar})$ the transfer function becomes

$$
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = (s_l, s_g, s_h, s_t[t_{tar} \mapsto \text{Value}(\text{emptyArray})], s_{ht}) \tag{4.25}
$$

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

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

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

Appending a value on a set of locations is done in the same manner as before, joining the existing array with a list-array for each location. Here however the value being appended is also added to the temporary variable $t_{tar}$, since this is what an append returns in PHP. For $n = array_a(h_{var}, t_{val}, t_{tar})$ the transfer function is defined as

$$
\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \texttt{let } l' = \text{HLoc}(\delta, n, 0)\texttt{in} \\
& \texttt{let } s'_t = s_t[t_{tar} \mapsto s_t(t_{val})]\texttt{in} \\
& \texttt{let } s'_h = s_h[ \\
& \quad l' \mapsto s_t(t_{val}), \\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \quad \texttt{let } (v_a, v_s, v_n, v_b, v_u) = s_h(l)\texttt{in} \\
& \quad l \mapsto (v_a \sqcup \text{ArrayList}(\{l'\}), v_s, v_n, v_b, v_u)] \\
& \texttt{in} \\
& (s_l, s_g, s'_h, s'_t, s_{ht})
\end{aligned} \tag{4.27}
$$

Appending value in the form of a set of locations to a set of locations 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. For $n = array_a(h_{var}, h_{val}, t_{tar})$ the transfer function becomes

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

The final array-append operation occurs when an array is appended and immediately thereafter accessed. This is the case in program 4.3. Here a new location, $l'$ is created and appended, in the same was as previously, to all possible locations. Since the lattice initializes new locations to Value(Null($\top$)) it is important, and sound, to set $l'$ to Value($\perp$) since the location will be joined with another array immediately after. For $n = array_a(h_{var}, h_{tar})$ the transfer function is defined as

$$
\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = & \texttt{let } l' = \text{HLoc}(\delta, n, 0)\texttt{in} \\
& \texttt{let } s'_{ht} = s_{ht}[h_{tar} \mapsto \{l'\}]\texttt{in} \\
& \texttt{let } s'_h = s_h[ \\
& \quad l' \mapsto \text{Value}(\perp), \\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \quad \texttt{let } (v_a, v_s, v_n, v_b, v_u) = s_h(l)\texttt{in} \\
& \quad l \mapsto (v_a \sqcup \text{ArrayList}(\{l'\}), v_s, v_n, v_b, v_u)] \\
& \texttt{in} \\
& (s_l, s_g, s'_h, s_t, s'_{ht})
\end{aligned} \tag{4.29}
$$

**Program 4.3** Array append before write

```
$a = [];
$a[]["foo"] = 42;
```

When writing or reading from an array, given a value, $v$, as key, the value must first be coerced to an array index. The easy approach would be to use the coercion function directly $c_{\text{Value,ArrayIndex}}(v)$, which coerces and then joins all factors. Another approach would be to coerce the factors and write to or join the corresponding value of the indicies individually. This last method will most likely involve at least one $\bot$ array index, which will become a problem. The problem becomes apparent when considering how values written to and read from map-arrays. Writing index $i$ with location set $L$ on some map-array, $a$, should ideally be done by joining the location set of each entry in $a$ with $L$ where $i$ is contained in the corresponding key. E.g. $a[\forall d \in dom(a) \wedge i \sqsubseteq d.d \mapsto a(d) \sqcup L]$. 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 $d \in dom(a)$ where $d \sqsubseteq i'$ or $i' \sqsubseteq d$, 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 is ensured. Therefor a third option is to only consider coerced factors, of the key value, that are not contained in other factors. These are the indices returned by the function $indices : \text{Value} \to \text{ArrayIndex}^*$

$$
\begin{aligned}
indices(v) = &\texttt{let } (v_a, v_s, v_n, v_b, v_u) = v \texttt{in} \\
&\texttt{let } I = \{c_{\text{Array,Index}}(v_a), \\
&\quad c_{\text{String,Index}}(v_s), \\
&\quad c_{\text{Number,Index}}(v_n), \\
&\quad c_{\text{Boolean,Index}}(v_b), \\
&\quad c_{\text{Null,Index}}(v_n)\}\texttt{in} \\
&I \setminus \{j | i, j \in I \wedge j \sqsubseteq i \wedge i \neq j\}
\end{aligned} \tag{4.30}
$$

Using the above function, we are able to generalize reading from an array as a function $readArray : \text{Value} \times \text{Value} \times \text{Heap} \to \mathcal{P}(L)$ 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. Reading from $\bot$ or emptyMap results in the empty set, since they contain no locations. Finally reading from $\top$ results in all possible locations, e.g. the top element of $\mathcal{P}(L)$.

$$
\begin{aligned}
\mathit{readArray}(v,k,h) = & \texttt{let } (v_a,v_s,v_n,v_b,v_u) = v \texttt{in} \\
& \texttt{match } v_a \\
& \quad \texttt{with } \top\text{: } \mathit{dom}(h) \\
& \quad \texttt{with } \text{ArrayList}(L)\text{: } L \\
& \quad \texttt{with } \text{ArrayMap(m)}\text{:} \\
& \quad \bigcup_{d \in \mathit{dom}(m) \wedge \exists i \in \mathit{indices}(k).i \sqsubseteq d \vee d \sqsubseteq i} m(d) \\
& \quad \texttt{with } \text{emptyArray: } \emptyset \\
& \quad \texttt{with } \bot\text{: } \emptyset
\end{aligned} \tag{4.31}
$$

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

$$
\begin{aligned}
\mathit{writeArray}(v,k,L) = & \texttt{let } (v_a,v_s,v_n,v_b,v_u) = v \texttt{in} \\
& \texttt{let } m = \text{ArrayMap}([\forall i \in \mathit{indices}(k).i \mapsto L])\texttt{in} \\
& (\texttt{match } v_a \\
& \quad \texttt{with } \text{ArrayList}(L')\text{:} \\
& \quad\quad \texttt{if } \exists i \in \mathit{indices}(k).i \texttt{ is } \text{String}(\_) \texttt{ then} \\
& \quad\quad\quad \text{ArrayMap}([\top \mapsto L']) \sqcup m \texttt{ else} \\
& \quad\quad\quad \text{ArrayList}(L \cup L') \\
& \quad \texttt{with } \text{ArrayMap}(m')\text{: } m' \sqcup m \\
& \quad \texttt{with } \text{emptyArray: :} \\
& \quad\quad \texttt{if } \exists i \in \mathit{indices}(k).i \texttt{ is } \text{String}(\_) \texttt{ then} \\
& \quad\quad\quad m \texttt{ else } \text{ArrayList(L)} \\
& \quad \texttt{with } \bot\text{: :} \\
& \quad\quad \texttt{if } \exists i \in \mathit{indices}(k).i \texttt{ is } \text{String}(\_) \texttt{ then} \\
& \quad\quad\quad m \texttt{ else } \text{ArrayList(L)} \\
& \quad \texttt{with } \top\text{: : } \top, v_s, v_n, v_b, v_u)
\end{aligned} \tag{4.32}
$$

With these functions in place, the transfer functions for the array-read and write nodes can be defined. For $n = \mathit{array}_r(t_{ar}, t_{key}, t_{tar})$ a value is read from an array in at temporary variable $t_{ar}$ as mentioned above. The query is always joined with Value(Null($\top$)) since an entry might not be set. The transfer function is defined as

$$
\begin{aligned}
f_{n,\delta}((s_l,s_g,s_h,s_t,s_{ht})) = & \texttt{let } L = \mathit{readArray}(s_t(t_{ar}), s_t(t_{key}), s_h)\texttt{in} \\
& \texttt{let } v = \text{Value}(\text{Null}(\top)) \sqcup s_h(L)\texttt{in} \\
& (s_l, s_g, s_h, s_t[t_{tar} \mapsto v], s_{ht})
\end{aligned} \tag{4.33}
$$

When reading the locations of an array index all arrays returning no locations must be updated for the same reasons as when the locations a variable is read. For $n = array_r(h_{var}, t_{key}, h_{tar})$ the transfer function becomes

$$
\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) =& \texttt{let } l = \text{HLoc}(\delta, n, 0)\texttt{in} \\
& \texttt{let } s'_{ht} = s_{ht}[h_{tar} \mapsto \\
& \quad \cup_{l' \in s_{ht}(h_{var})} \mathit{cardCheck}( \\
& \qquad \mathit{readArray}(s_h(l'), s_t(t_{key}), s_h), l)] \\
& \texttt{in} \\
& \texttt{let } s'_h = \\
& \quad s_h[\forall l' \in s_{ht}(h_{var}) \\
& \qquad\quad \wedge \mathit{readArray}(s_h(l'), s_t(t_{key}), s_h) = \emptyset. \\
& \qquad l' \mapsto \mathit{writeArray}(s_h(l'), s_t(t_{key}), \{l\})] \\
& \texttt{in} \\
& (s_l, s_g, s'_h, s_t, s'_{ht})
\end{aligned} \tag{4.34}
$$

where

$$
\mathit{cardCheck}(L, l) = \texttt{if } L = \emptyset \texttt{ then } \{l\} \texttt{ else } L \tag{4.35}
$$

Writing to an array at temporary variable $t_{ar}$ is done easily using the *writeArray* function. Here the value, at temporary variable $t_{val}$ is stored in the heap at a new location, which is consequently written to the existing array. For $n = array_w(t_{key}, t_{val}, t_{ar})$ the transfer function is defined as

$$
\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) =& \texttt{let } l' = \text{HLoc}(\delta, n, 0)\texttt{in} \\
& \texttt{let } v' = \texttt{in} \\
& (s_l, s_g, s_h[l' \mapsto s_t(t_{val})], \\
& \quad s_t[t_{ar} \mapsto \mathit{writeArray}(s_t(t_{ar}), s_t(t_{key}), \{l'\})], s_{ht})
\end{aligned} \tag{4.36}
$$

Writing to a set of locations is more of the same. For $n = array_w(h_{var}, t_{key}, t_{val}, t_{tar})$ the transfer function becomes

$$
\begin{aligned}
f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) =& \texttt{let } l' = \text{HLoc}(\delta, n, 0)\texttt{in} \\
& \texttt{let } s'_t = s_t[t_{tar} \mapsto s_t(t_{val})]\texttt{in} \\
& \texttt{let } s'_h = s_h[ \\
& \quad l' \mapsto s_t(t_{val}), \\
& \quad \forall l \in s_{ht}(h_{var}). \\
& \quad l \mapsto \mathit{writeArray}(s_h(l), s_t(t_{key}), \{l'\})] \\
& \texttt{in} \\
& (s_l, s_g, s'_h, s'_t, s_{ht})
\end{aligned} \tag{4.37}
$$

Writing a value, as a set of locations, to an array, also as a set of locations, is done in the same manner as the previous function. Thus for $n =$

$array_w(h_{var}, t_{key}, h_{val}, t_{tar})$ the transfer function is defined as

$$f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = \texttt{let } s'_t = s_t[t_{tar} \mapsto s_t(s_h(s_{ht}(h_{val})))] \texttt{in}$$
$$\texttt{let } s'_h = s_h[$$
$$\forall l \in s_{ht}(h_{var}).$$
$$l \mapsto writeArray(s_h(l), s_t(t_{key}), s_{ht}(h_{val}))]$$
$$\texttt{in}$$
$$(s_l, s_g, s'_h, s'_t, s_{ht}) \tag{4.38}$$

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. As mentioned before, no strong updates can be performed, while maintaining soundness, on either variables or arrays. This follows from how PHP performs deep copy, which is described in section 2.3. By copying the references PHP opens the possibility for the modification of a deep-copied array through another variable/array, with no reference assignment from the array. In order to be sound, the analysis has to assume that no arrays are deep copied, but instead shares the internal references of the original array. From this follows that no strong-updates can be performed on any location, because it might result in an update of an array which in practice never share the location of the variable. This issue is illustrated in program 4.4. If strong updates were allowed, updating `$c` in the last line would result in `$a` rightfully and `$b` wrongfully being updated to the list containing the number two, `[2]`, since the two arrays share the same internal locations. By only prefroming weak updates, the two arrays becomes a list of UIntNumber which is sound.

**Program 4.4**

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

### 4.4.4 Function calls

The transfer functions for function call, *call* and *result* nodes, differs from the other functions. For $n = call_{fn}(c_1, \ldots, c_n)$ the transfer function $f_{n,\delta}$ : AnalysisLattice → AnalysisLattice sets up the local scope for the function body of $fn$. The scope is initially empty with exception of the arguments being set. Setting the arguments is done by reference or by value, which is indicated by the call arguments, $c_1, \ldots, c_n$, being a THVar or a TVar respectively. If the argument is passed by reference, then the corresponding argument is set, in the local scope, to point at the provided heap locations. If the argument is passed by value, then the argument is pointing to a newly created heap location, which in turns points to the value. The second case is, including the choice of start-lattice, is the reason for HLoc being defined as a product of context, node, and a natural number. Without the third factor all, value-passed, arguments would be written to the same heap-location. This is avoided by setting the number in the heap location to the position of the argument in question. The global scope and heap is preserved in the new scope, while the temporary maps both are *emptied*. The transfer function is defined as

$$
\begin{aligned}
f_{n,\delta}(l) = & \texttt{let } \delta' = \mathit{addCallNode}(\delta, n)\texttt{in} \\
& \texttt{let } (\_, s_g, s_h, s_t, \_) = l[\delta]\texttt{in} \\
& \texttt{let } s_l = [\forall(v,i) \in \mathit{args}(fn).v \mapsto \\
& \quad \texttt{match } c_i \\
& \quad\quad \texttt{with } \text{TVar: } \{\text{HLoc}(\delta, \mathit{startNode}(\mathit{fn}), i)\} \\
& \quad\quad \texttt{with } \text{THVar: } s_h(c_i)] \\
& \texttt{in} \\
& \texttt{let } s'_h = [\forall(\_, i) \in \mathit{args}(fn).\text{HLoc}(\delta, \mathit{startNode}(\mathit{fn}), i) \mapsto \\
& \quad \texttt{match } c_i \\
& \quad\quad \texttt{with } \text{TVar: } s_t(c_i) \\
& \quad\quad \texttt{with } \text{THVar: } s_h(\text{HLoc}(\delta, \mathit{startNode}(\mathit{fn}), i))] \\
& \texttt{in} \\
& (s_l, s_g, s'_h, [], [])
\end{aligned} \tag{4.39}
$$

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

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

$$
\begin{aligned}
f_{n,\delta_{call},\delta_{exit}}(l_{call}, l_{exit}) = & \texttt{let}\ \ \mathit{exit}(c_1, \cdots, c_n) = \mathit{exitNode}(fn)\texttt{in} \\
& \texttt{let}\ (s_l, \_, \_, s_t, s_{ht}) = l_{call}(\delta_{call})\texttt{in} \\
& \texttt{let}\ (\_, s_g, s_h, s'_t, s'_{ht}) = l_{exit}(\delta_{exit})\texttt{in} \\
& \texttt{let}\ v = \\
& \quad \bigsqcup_{0<i\leq n} \texttt{match}\ c_i \\
& \qquad\quad \texttt{with}\ TVar\colon\ s_t(c_i) \\
& \qquad\quad \texttt{with}\ THVar\colon\ s_h(s_{ht}(c_i)) \\
& \texttt{in} \\
& (s_l, s_g, s_h, s_t[t_{val} \mapsto v], s_{ht})
\end{aligned}
\tag{4.40}
$$

for $n = \mathit{result}_{call_{fn}}(t_{val})$ or for $n = \mathit{result}_{call_{fn}}(h_{val})$ as

$$
\begin{aligned}
f_{n,\delta_{call},\delta_{exit}}(l_{call}, l_{exit}) = & \texttt{let}\ \ \mathit{exit}(c_1, \cdots, c_n) = \mathit{exitNode}(fn)\texttt{in} \\
& \texttt{let}\ (s_l, \_, \_, s_t, s_{ht}) = l_{call}(\delta_{call})\texttt{in} \\
& \texttt{let}\ (\_, s_g, s_h, s'_t, s'_{ht}) = l_{exit}(\delta_{exit})\texttt{in} \\
& \texttt{let}\ L = \\
& \quad \bigcup_{0<i\leq n} \texttt{match}\ c_i \\
& \qquad\quad \texttt{with}\ TVar\colon\ \{\mathrm{HLoc}(n, \delta_{call}, i)\} \\
& \qquad\quad \texttt{with}\ THVar\colon\ s_{ht}(c_i) \\
& \texttt{in} \\
& \texttt{let}\ s'_h = s_h[\forall 0 < i \leq n.\mathrm{HLoc}(n, \delta_{call}, i) \mapsto \\
& \quad \texttt{match}\ c_i \\
& \qquad \texttt{with}\ TVar\colon\ s_t(c_i) \\
& \qquad \texttt{with}\ THVar\colon\ s_h(\mathrm{HLoc}(n, \delta_{call}, i))] \\
& \texttt{in} \\
& (s_l, s_g, s'_h, s_t, s_{ht}[h_{val} \mapsto L])
\end{aligned}
\tag{4.41}
$$

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

### 4.4.5 Other transfer functions

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

$$f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = (s_l, s_g, s_h, s_t[t_{tar} \mapsto value(c)], s_{ht}) \tag{4.42}$$

and finally 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 current scope is empty, them no modifications are made to the input lattice. The transfer function is defined as

$$\begin{aligned} f_{n,\delta}((s_l, s_g, s_h, s_t, s_{ht})) = &\texttt{match } \delta \\ &\quad \texttt{with } \Lambda\text{: } (s_l, s_g, s_h, s_t, s_{ht}) \\ &\quad \texttt{with } \_\text{: } (s_l[\forall 0 \leq i \leq n.v_i \mapsto s_g(v_i)], s_g, s_h, s_t, s_{ht}) \end{aligned} \tag{4.43}$$

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

### 4.4.6 Applying transfer functions

For control flow nodes with a single incoming edge applying the transfer function is simple. The lattice associated with the other node of the incoming edge has the transfer function applied directly. If multiple incoming edges exist the corresponding lattices have to be combined by their least-upper-bound. The least-upper-bound can be calculated either before or after applying the transfer function. Applying the transfer function before finding the least-upper-bound requires applying the transfer function multiple times and while this increases the precision of the analysis it decreases the performance. Because of the small size of the cases contained in the case study the precision and performance impact of choosing either method will not be noticeable. The choice fell upon applying transfer functions and then finding the least-upper-bound due to implementation details.

**►Write about how we read array entries, e.g. containment of keys and stuff◄**

## 4.5 Coercion

**►description first, definition afterwards. More description of interesting cases of coercion◄** In order to mimic the ability of coercing values the following functions

$$c_{\alpha,\beta} : \alpha \to \beta$$

where $\alpha \in \{\text{Value}, \text{Array}, \text{Number}, \text{String}, \text{Null}, \text{Boolean}\}$ and $\beta \in \{\text{Number}, \text{Value}, \text{String}, \text{Index}, \text{Boolean}\}$ are defined as

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

$$c_{\text{Value},\alpha}((v_1, v_2, v_3, v_4, v_5)) = c_{\text{Array},\alpha}(v_1) \sqcup c_{\text{String},\alpha}(v_2) \sqcup c_{\text{Number},\alpha}(v_3) \sqcup c_{\text{Boolean},\alpha}(v_4) \sqcup c_{\text{Null},\alpha}(v_5)$$

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

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

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

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

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

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

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

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

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

Here *int*, *num*, and *string* are creating an integer, number, and string respectively.

A few things are worth noticing about coercion in PHP.

- Any array coerced to a string will result in the string literal "Array"
- Null is coerced to the empty string
- Both the empty string and the string literal "0" are coerced to boolean `false`
- Boolean `false` is coerced to the empty string
- An empty array is coerced to boolean `false`

.

## 4.6 Abstract evaluation

This section lists the supported binary and unary operators and the abstract evaluation of those operators for the analysis lattice. Insight on the abstract evaluation can be found in this section. Appendix B provides definition tables for all abstract operators for reference.

The binary and unary operations supported are listed in table 4.1 and 4.2 respectively. Numeric, boolean and string operators can be generalized to values by using the coercion function from the previous section. E.g. given values $x$ and $y$

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

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

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

where $\circ : \alpha \to \alpha$.

**►examples◄**

In table 4.1 logical `AND` and `OR` operations 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 definitions use the following variables:

$$\begin{aligned} &x, y \in \mathbb{Z} \wedge x, y > 0 \\ &a, b \in \mathbb{R} \wedge (a, b < 0 \vee a, b \notin \mathbb{Z}) \end{aligned}$$

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

For division and modulo operations an error can occur trying to divide by 0. PHP handles division by 0 by returning the boolean `false` value instead of a number. This is the reason that division and modulo operators are functions from numbers to a value as opposed to the rest of the numeric operators. The shorthand $f$ is used for the boolean `false` value to keep the table smaller. Only the uInt part of the number lattice can contain 0 which restricts the amount of possible `false` returns for division. However as seen in table 4.4 the amount of possible `false` values are high for the modulus operator. This is because PHP handles modulus for decimal numbers by removing the decimal part which means $-0.5 \Rightarrow 0$ and $0.8 \Rightarrow 0$. The result of the modulus operator is always an integer and the sign is dictated by the sign of the left-hand operand

| 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`<br>`x AND y` | Logical and | Boolean × Boolean → Boolean |
| `x \|\| y`<br>`x OR y` | Logical or | |
| `x XOR y` | Exclusive or | |
| `x . y` | String concatenation | String × String → String |

Table 4.1: Binary operations

| Operator | Name | Signature |
|---|---|---|
| `! x` | Negation | Boolean → Boolean |
| `- x` | Decrement | Number → Number |
| `+ + x` | Increment | |
| `x - -` | Decrement | |
| `x + +` | Increment | |
| `- x` | Unary Minus | |

Table 4.2: Unary operations

| $-$ | $\bot$ | $0$ | $y$ | 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 4.3: Abstract Subtraction

| % | $\bot$ | 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 | $0 \sqcup f$ | $0 \sqcup f$ |
| $x$ | $\bot$ | $f$ | $x\%y$ | 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 $\sqcup f$ |
| $1 > a > -1$ | $\bot$ | $f$ | $a\%y$ | 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 4.4: Abstract Modulus

which can be seen in the table by the $\top$-element column not consisting solely of $\top$-element results.

Comparison operations are defined directly on values since different types can be compared with each other courtesy of type translation. PHP has an ordered definition of how values are compared depending on their type which can be seen in table 4.5.

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

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

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

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

## 4.7 The Monotone Framework

In order to perform a data flow analysis on a P0 program, the embellished monotone framework is used ►ref◄. Given a control flow graph, $G = (V, E, s, t)$, an instance of the monotone framework, $(L, \mathcal{F}, F, E', \iota, f)$, with data flow equations, $A : \mathcal{N} \times \Delta \to L$,

$$A_\bullet((n,\delta)) = \begin{cases} f_{(n,\delta)}(A_\circ((\delta,c)), A_\circ((n,\delta))) & \text{if } n = return_c(\_) \\ f_{(n,\delta)}(A_\circ(l)) & \text{else} \end{cases} \tag{4.44}$$

$$A_\circ(l) = \bigsqcup\{A_\bullet(l')|(l',l) \in F\} \sqcup \iota^l_{E'} \tag{4.45}$$

where

$$\iota^l_{E'} = \begin{cases} \iota & \text{if } l \in E' \\ \perp & \text{else} \end{cases} \tag{4.46}$$

can be derived. Here $L = \text{AnalysisLattice}$, $\mathcal{F}$ is the set of transfer functions in in the analysis, closed under composition, $f$ is defined throughout section 4.4, $F : (\mathcal{N} \times \Delta) \times (\mathcal{N} \times \Delta)$ is defined as

$$\begin{aligned} F =& \{((n,\delta),(n',\delta'))|(n,n') \in E \wedge \delta \in \Delta \\ & \wedge \mathit{validSuccessor}(n,n') \\ & \wedge \delta' = \mathit{nextC}(n,\delta)\} \end{aligned} \tag{4.47}$$

where

$$\mathit{nextC}(n,\delta) = \begin{cases} \delta' & \text{if } n = \mathit{exit}(\_) \text{ and } \delta = (\delta', c) \\ (\delta, n) & \text{if } n = \mathit{call}(\_) \\ \delta & \text{else} \end{cases} \tag{4.48}$$

and *validSuccessor* is as defined in definition 7.

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

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

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

## 4.8 Worklist algorithm

In order to solve the data-flow equation of the monotone framework, the above lattice, control-flow-graph, and transfer functions has been implemented in approximately 8100 lines of Java code, as a plug-in in the Intellij IDEA (Ultimate edition) development environment, by JetBrains►**ref**◄. This IDE supports multiple languages, such as Java, Python, C, C++, C#, Ruby, and PHP, with tools for re-factoring, type-checking, a vast library of plug-ins, developed by JetBrains or the JetBrains community, etc.

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

The lattices are defined by interfaces and the elements are implemented as immutable data structures of these interfaces. In the name of efficiency the domain of map-lattice elements (`MapLatticeElement`) are defined as the indices of modified entries, not including the values defaulted to 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* does also entail that when updating the entry of a $\top$-array, only the variables initialized are effected, which is sound and yields a more precise model.

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

- When ???
- **►List errors◄**

Solving the data-flow equations are performed by an implementation of the worklist algorithm.

**Algorithm 1** Worklist algorithm

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

$I = [\forall \delta \in \Delta, n \in \mathcal{N}.(n, \delta) \mapsto \bot]$

$I[(s, \Lambda) \mapsto \iota]$

$W = [f \in F | f = ((s, \Lambda), \_)]$

**while** $W \neq \emptyset$ **do**

  $(l_1, l_2) = W.takeFirst()$

  **if** $f_{l_1}(l_1) \not\sqsubseteq I[l_2]$ **then**

    $I[l_2 \mapsto f_{l_1}(l_1)]$

    $W.append([f \in F | f = (l_2, \_)])$

  **end if**

**end while**

**►Library functions◄ ►write about the implementation◄**
**►look and feel◄**
**►Short cuts?◄**

# Chapter 5

# Case Study

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

**Program 5.1** Array used as a map

```
$bob
```

# Chapter 6

# Related work

## 6.1 Dynamic features

In this thesis a lot of the dynamic features of PHP have not been covered. ▶**describe related work that cover dynamic features**◀.

## 6.2 Evolution of Dynamic Feature Usage in PHP

Purpose: examine dynamic feature usage in WordPress and MediaWiki.

Method: static analysis of occurences relative to total lines of code

Important points: Differentiating between test code, maintenance code and actual end-user visible code. E.g. test code use more dynamic features in MediaWiki.

Conclusion: A decrease in eval and create_function usage as language features are added that can replace typical use cases for those features. Increase in magic methods use and variable property use. [**?**]

## 6.3 Type Analysis for JavaScript

▶**write summary**◀ [**?**]

## 6.4 WeVerca

▶**write summary**◀ Using PHP 4 [**?**]

## 6.5 A static analyser for finding dynamic programming errors

Purpose: finding pointer errors, memory leaks and resource leaks

Method: Simulating paths and modelling functions with guards and constraints

Important points: keeping exact values as much as possible, using predicates when exact values are unknown [**?**]

## 6.6 Static Approximation of Dynamically Generated Web Pages

**▶write summary◀** [?]

## 6.7 Static Detection of Cross-Site Scripting Vulnerabilities

**▶write summary◀** [?]

## 6.8 Static Detection of Security Vulnerabilities in Scripting Languages

**▶write summary◀** [?]

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

**▶write summary◀** [?]

## 6.10 Sound and Precise Analysis of Web Applications for Injection Vulnerabilities

**▶write summary◀** [?]

## 6.11 Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities

**▶write summary◀** Using PHP 4 [?]

## 6.12 An Empirical Study of PHP Feature Usage

Purpose: which features are used in real applications

Method: corpus of most popular PHP applications (framework, e-commerce, learning platform, forum software), analyse feature usage and distribution

Conclusions: eval is used in real contexts

Application release dates range from 2010 to 2012. The analyser code is available so it might be interesting to run the analysis on newer versions of the same applications see how results have changed. [?]

## 6.13 Alias Analysis for Object-Oriented Programs

►**write summary**◄ [?]

## 6.14 Two Approaches to Interprocedural Data Flow Analysis

►**write summary**◄ [?]

## 6.15 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 [?]

## 6.16 Blended Analysis for Performance Understanding of Framework-based Applications

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

# Chapter 7

# Conclusion

►. . . ◄

# Chapter 8

# Future Work

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

**►. . . ◄**

# Chapter 9

# Work schedule

| Week | Task |
|---|---|
| 12 | |
| 13 | dynamic analysis finish |
| 14 | static analysis research |
| 15 | static analysis |
| 16 | static analysis + prototype implementation |
| 17 | static analysis + prototype implementation |
| 18 | implementation of static analysis |
| 19 | implementation of static analysis |
| 20 | buffer |
| 21 | finishing touch on analyses |
| 22 | writing |
| 23 | writing |
| 24 | proof-reading |
| 25 | hand-in |

►...◄

# Appendix A

# Basic Definitions

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

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

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

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

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

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 12.** 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 13.** 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 operators for 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

| $\oplus$ | $\bot$ | 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 (Equal) and Greater Than (Equal) for Strings, replace $\oplus$ with the respective operators

| $==$ | $\bot$ | 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 for 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 for Numbers

| $<$ | $\bot$ | y | uInt | b $<$ 0 | b | notUInt | $\top$ |
|---|---|---|---|---|---|---|---|
| $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| x | $\bot$ | x $<$ y | $\top$ | f | x $<$ b | $\top$ | $\top$ |
| uInt | $\bot$ | $\top$ | $\top$ | f | $\top$ | $\top$ | $\top$ |
| a $<$ 0 | $\bot$ | t | t | a $<$ b | a $<$ b | $\top$ | $\top$ |
| a | $\bot$ | a $<$ y | $\top$ | a $<$ b | a $<$ 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 for Numbers, the table for Less Than/Equal corresponds to this one

| $>$ | $\bot$ | y | uInt | b $<$ 0 | b | notUInt | $\top$ |
|---|---|---|---|---|---|---|---|
| $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| x | $\bot$ | x $>$ y | $\top$ | t | x $>$ b | $\top$ | $\top$ |
| uInt | $\bot$ | $\top$ | $\top$ | t | $\top$ | $\top$ | $\top$ |
| a $<$ 0 | $\bot$ | f | f | a $>$ b | a $>$ b | $\top$ | $\top$ |
| a | $\bot$ | a $>$ y | $\top$ | a $>$ b | a $>$ 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 for Numbers, the table for Greater Than/Equal corresponds to this one

| `&&` | $\bot$ | 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 Operators for Booleans