A Constraint-Based Method for Flow-Sensitive Static Type Analysis of PHP Using the Rascal Meta-Programming Platform,

by

Apil Tamang.

July, 2015.

Director of Thesis: Mark Hills, PHD.

Major Department: Computer Science, ECU.

PHP is a dynamically typed language, and is very popular among developers for building websites. The dynamic features of PHP allow for many kinds of type related errors to be made by developers. Since PHP does not feature a static type checker, these errors cannot be detected until runtime. When such an error is encountered, it often results in the application to crash, thus possibly loosing valuable computational time and effort. Worse yet, the PHP compiler may use type coercion to convert an incompatible data to the required type, and in doing so, introduce an extremely hard-to-trace application bug.

In this research, we address this problem by building a type-checker that performs a flow-sensitive type-inference of PHP expressions. The analysis is based on a constraints-based algorithm, and utilizes the program control flow-graph for visiting all reachable nodes in the application. The type-checker uses the inferred types for performing many type-related analyses. Results of the analyses are displayed to the developers for locating any constructs that may result in a possible application failure. By continually using this feedback, the tool aims to help PHP developers build more robust PHP applications.
A Constraint-Based Method for Flow-Sensitive Static Type Analysis of PHP Using the Rascal Meta-Programming Platform.

by

Apil Tamang

A thesis submitted to the Graduate Faculty of East Carolina University in partial fulfillment of the requirements for the Degree of Master of Science in Software Engineering

July, 2015
A Constraint-Based Method for Flow-Sensitive Type Analysis of PHP Using the Rascal Meta-
Programming Platform,

by

Apil Tamang.

APPROVED BY:

DIRECTOR OF
THESIS:

Mark Hills, PhD.

COMMITTEE MEMBER:

M. Nasseh Tabrizi, PhD.

COMMITTEE MEMBER:

Sergiy Vilkomir, PhD.

CHAIR OF THE DEPARTMENT
OF COMPUTER
SCIENCE

Karl R. Abrahamson, PhD.

DEAN OF THE
GRADUATE
SCHOOL

Paul J. Gumperline, PhD.
DEDICATION

To my parents, friends, and wife, Pratima Karki, without whose support I could not have done it.
ACKNOWLEDGEMENTS

First and foremost, this work would not have seen the light of day without the extended support of Dr. Mark Hills. Thanks for showing an incredible patience with the progress of this work. Much indebtedness also extend towards Dr. Nasseh Tabrizi for giving me the opportunity to pursue my Master's work and to the entire East Carolina University (ECU) family for providing the necessary academic environment for seeing through this endeavor. As always, my family and friends have no less stake on the success of this project. Lastly, once again to Pratima for being such a wonderful companion through this, and for many more journeys to come.
# TABLE OF CONTENTS

**LIST OF TABLES** .......................................................... viii

**LIST OF FIGURES** ........................................................ ix

**Chapter 1**  **The PHP Language** ............................................ 1
  1.1 A Short Overview .................................................. 1
  1.2 Variables and expressions ............................................ 2
  1.3 Include and Require Statements ....................................... 4
  1.4 Dynamic Nature of PHP Functions ..................................... 5
  1.5 Classes and Objects ................................................ 7
    1.5.1 Static and instance class members ................................ 10
    1.5.2 Object Aliasing ................................................ 11
    1.5.3 Other sources of errors .......................................... 13
  1.6 Conclusion ........................................................ 14

**Chapter 2**  **Rascal** .................................................... 15
  2.1 Introduction .................................................... 15
  2.2 Built-In Type System ................................................ 16
    2.2.1 The *loc* type .............................................. 16
    2.2.2 Rascal collection primitives ................................. 18
    2.2.3 Algebraic Data Types (ADTs) .................................. 19
  2.3 Program Constituents ............................................. 21
    2.3.1 Import ....................................................... 22
    2.3.2 Syntax definition ............................................ 22
    2.3.3 Variable ..................................................... 22
    2.3.4 Function ..................................................... 23
    2.3.5 Alias ........................................................ 25
    2.3.6 Annotation .................................................. 25
  2.4 Patterns and Pattern-Matching .................................... 26
    2.4.1 Patterns as control expressions .............................. 27
    2.4.2 Patterns as function parameters .............................. 29
  2.5 Conclusion ....................................................... 30

**Chapter 3**  **Type Analysis for PHP** ........................................... 32
  3.1 Analysis of Methodoloy ............................................ 32
    3.1.1 Type Systems ................................................ 32
    3.1.2 Related Work ................................................ 34
    3.1.3 Type Language ................................................. 36
    3.1.4 Method Overview ............................................. 38
    3.1.5 Type unification ............................................. 39
    3.1.6 Analysis of constraints ................................. 42
  3.2 Constraints Generation ........................................... 46
LIST OF TABLES

Table 2.1   Abstract pattern syntax in Rascal .............................. 27
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Syntax for Include and Require statements in PHP</td>
</tr>
<tr>
<td>1.2</td>
<td>Example of a PHP function definition taking variable number of parameters</td>
</tr>
<tr>
<td>1.3</td>
<td>Example of a PHP function definition potentially returning a null type.</td>
</tr>
<tr>
<td>1.4</td>
<td>Example of a class definition in PHP</td>
</tr>
<tr>
<td>1.5</td>
<td>Examples of using static class members in PHP</td>
</tr>
<tr>
<td>1.6</td>
<td>Object referencing in PHP</td>
</tr>
<tr>
<td>1.7</td>
<td>Calling a method invocation on a non-object variable.</td>
</tr>
<tr>
<td>2.1</td>
<td>Rascal type system lattice [Rscb]</td>
</tr>
<tr>
<td>2.2</td>
<td>Top-level Statements in Rascal Modules [Rsca]</td>
</tr>
<tr>
<td>2.3</td>
<td>Syntax for Import Statement in Rascal</td>
</tr>
<tr>
<td>2.4</td>
<td>Variable Typing in Rascal</td>
</tr>
<tr>
<td>2.5</td>
<td>Syntax for Functions in Rascal</td>
</tr>
<tr>
<td>2.6</td>
<td>Definition of a Function Returning a Tuple Type</td>
</tr>
<tr>
<td>2.7</td>
<td>Using Alias to Define New Reference for a Type in Rascal</td>
</tr>
<tr>
<td>2.8</td>
<td>Simple Example for Using Annotation in Rascal</td>
</tr>
<tr>
<td>2.9</td>
<td>Pattern matching usages in Rascal</td>
</tr>
<tr>
<td>2.10</td>
<td>Example of using patterns as formal function parameters</td>
</tr>
<tr>
<td>2.11</td>
<td>More pattern-based function definitions from Fig. 2.10.</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of a case when a php variable may have more than one type</td>
</tr>
<tr>
<td>3.2</td>
<td>Base set of concrete types in the Analysis</td>
</tr>
<tr>
<td>3.3</td>
<td>Non-terminating type analysis iteration without Type unification</td>
</tr>
<tr>
<td>3.4</td>
<td>Type lattice resulting from using the $\n_4$ operator.</td>
</tr>
<tr>
<td>3.5</td>
<td>Definition of the widening ($\n$) operator for an integer value $k$</td>
</tr>
<tr>
<td>3.6</td>
<td>Some examples of using the $\n_4$ operator to join two type sets</td>
</tr>
<tr>
<td>3.7</td>
<td>Some examples to illustrate array depth evaluations</td>
</tr>
<tr>
<td>3.8</td>
<td>Sequence of type sets evaluated for $a$ using the $\n_4$ operator</td>
</tr>
<tr>
<td>3.9</td>
<td>High level overview of the operations</td>
</tr>
<tr>
<td>3.10</td>
<td>Constraints used in this analysis</td>
</tr>
<tr>
<td>3.11</td>
<td>Constraints generated by literals in PHP</td>
</tr>
<tr>
<td>3.12</td>
<td>Constraints generated by basic expressions in PHP</td>
</tr>
<tr>
<td>3.13</td>
<td>Constraints generated by basic operators in PHP</td>
</tr>
<tr>
<td>3.14</td>
<td>Constraints generated by basic statements in PHP</td>
</tr>
<tr>
<td>3.15</td>
<td>Constraints generated by self-assignment statements in PHP</td>
</tr>
<tr>
<td>3.16</td>
<td>Constraints generated by control statements in PHP</td>
</tr>
<tr>
<td>3.17</td>
<td>Constraints generated by looping structures in PHP</td>
</tr>
<tr>
<td>3.18</td>
<td>Propagate type information through a php script</td>
</tr>
<tr>
<td>3.19</td>
<td>Outline of the transfer_type function</td>
</tr>
<tr>
<td>3.20</td>
<td>Outline of the resolve_dependencies function</td>
</tr>
<tr>
<td>3.21</td>
<td>Outline of the resolve_flow function</td>
</tr>
<tr>
<td>3.22</td>
<td>Outline of the get concrete type root function</td>
</tr>
</tbody>
</table>
Figure A.5  (...Continued Fig. A.4) Typed flow model of the script given in Fig A.1. . . . . . . 117
CHAPTER

1

THE PHP LANGUAGE

1.1 A Short Overview

PHP is one of the most widely used server-side scripting languages. It has been used to create several popular websites and website frameworks. As of May 2015, PHP powers 81.9% of all the websites whose server-side language can be determined [W3t], features the 5th largest number of active repositories on Github [Git] and is ranked 9th on the TIOBE programming community index [Tio].

The popularity of the language stems from its relatively low learning curve. Enthusiasts can get started on building production ready web systems with little overhead of server management and configuration tasks. The language features a powerful built in server and exposes a rich set of built in functions and an environment conducive to developing and designing websites. The language also features a minimally enforced type system for program variables. This enables developers to rapidly
1.2. VARIABLES AND EXPRESSIONS

recent releases of PHP support a fully functional object-oriented programming paradigm with support for object-oriented features such as multi-level inheritance, late static binding, and duck-typing.

PHP is an imperative programming language[Seb96] and features most standard procedural language constructs such as variables, statements, expressions, and operators. Programs in PHP are sequences of statements separated by a semicolon. A statement is usually comprised of one or more expressions. Statements may be from a wide variety of types, each one of which serves a unique task.

An expression is a unit of the code that can be evaluated to produce a value. There are two basic expressions: literals, and variables. Literal values evaluate to themselves, while variables evaluate to the values they store. More complex expressions are formed by combining base expressions with different operators and functions.

PHP supports a rich array of operators. Based on the number of expressions that an operator takes, they can be broadly classified into the following classes. The unary operators are a class of operators that operate on a single expression. The binary operators operate on two expressions and comprise the majority of the PHP’s operators. PHP also supports a class of special operators such as the scope resolution operator (::), the execution operator ('), and the ternary operator (cond ? action1 : action2).

And lastly, PHP programs may also consist of classes and functions declarations. Classes provide a way to define problem-specific data types in the application. Functions provide reusable application components that can be used to perform certain repeatable tasks. They are both core components of modern object-oriented languages and are used extensively for developing maintainable and scalable applications. We will review some more details of these language components in the rest of this chapter.

### 1.2 Variables and expressions

PHP is a dynamically typed language [Seb96; Tra09]. This means that the types of variables do not have to be specifically declared before they are used. Variable names must start with a $ sign followed by an identifier. PHP variables can be defined at any point in the program, and may also be redefined.
1.2. VARIABLES AND EXPRESSIONS  CHAPTER 1. THE PHP LANGUAGE

to a different data type later in the script. Additionally, a variable can be unset in the application at any location. This will cause all information in the variable to be lost.

To add to the dynamic nature of PHP variables, the PHP engine also dynamically coerces variable data to an expected type when required. For example, if a program variable $var is assigned to a string value as below,

$var = "12";

then using the arithmetic increment operator (++) on $var will result in a numerical value of 13. This is because the PHP engine coerces the value in $var to a numerical value as a result of using it ($var) in the context of an arithmetic operator (++). Now instead, if $var was defined as the following:

$var = "12a";

then applying the increment operator would return the string "12b". This happened because PHP follow's Perl's convention when applying C-like pre/post increment and decrement operators on individual characters. That means that the character next to a in the alphabet, i.e. b is evaluated and returned when evaluating the expression, ++ $var.

This process of internal type-casting is not only restricted to the above situations. Strings may also be used in the context of boolean expressions. For e.g. the following expression,

if ($var)
...

is perfectly valid. PHP evaluates all strings, with the exception of an empty string and the string: "0" as true expressions. Also, all numerical values other than 0 are evaluated as true expressions. The same applies for any resource, non-empty arrays, and PHP objects. This pattern is quite frequently used by PHP developers, and it is mostly a matter of choice whether to treat them as logical errors in the script. However, we believe that it is always a better practice to include explicit boolean
expressions whenever demanded, e.g.

\[
if( (mysql\_connection != null) \&\& (my\_string == "port2001") )\ldots
\]

The examples above highlight an important feature of the PHP language. It demonstrates that regardless of what a program variable may contain in the context of an expression, the PHP compiler will do its best not to fail. The PHP engine features a complex system of dynamic type-coercion rules that, almost always, enable it to evaluate something out of the given expression. These type-casting rules may often yield non-intuitive results, and introduce extremely hard-to-trace bugs. When the bug(s) finally do cause the application to crash, or yield wrong results, much loss in terms of computational time or other resources may have occurred.

Thus, a big part of our work is primarily directed by the notion that a variable’s content must conform to type of value expected in an expression of interest. When this conformance is broken, it is almost always an indication of an overlooked bug. The dynamic nature of PHP variables, i.e. situations where they can be reassigned to values of different types, unset, and even conditionally initialized, lead to a plethora of ways in which type-coercion bugs can be introduced in an application. The tool developed in this work is capable of detecting errors of this kind. In the last chapter, we give many examples of our tool capturing these errors.

### 1.3 Include and Require Statements

The include and require statements have very similar functionality. They are both used to copy any code or markup text that may appear in one file into another. The content is copied from the included file into the including file before the server executes any code in the script. This provides a convenient way for developers to re-use materials across several scripts. The only difference between these two statements is that if the PHP engine fails to find the file declared using the `include` keyword, the compiler issues a warning and execution of the script continues. If the PHP compiler fails to find the file declared using the `require` keyword, the compiler halts execution and throws a fatal error. The syntax for this statement is given in Fig. 1.1.

In PHP, it is also possible to provide an arbitrary expression for the path of a file to be included. The actual path of the file is then dynamically inserted during runtime. Although we will not deal
with this scenario in this work, we reference the work in [Hil14b] that attempts to statically resolve such include expressions. In the broader picture, this serves to illustrate one of the many dynamic features in PHP that makes type analysis of PHP programs so much more difficult.

```
include/require 'script_name';
```

**Figure 1.1** Syntax for Include and Require statements in PHP

### 1.4 Dynamic Nature of PHP Functions

PHP function definitions and calls are other language components that we've focused a significant portion of our effort on. They are interesting for type-analysis because they exhibit many dynamic features. For instance, a PHP function definition does not include any return type declaration. To that effect, the function can return one of multiple types of values through various locations in the function body. In addition to the dynamic nature of return types, the formal parameters of the functions are also not typed. Consequently, these parameters are bound to the types of values that are passed as arguments during a function call.

PHP allows a function to be called with a non-strict number of arguments. If the number of supplied arguments does not equal the number of declared parameters in the function definition, then PHP assigns null values to parameters that cannot be matched with an argument. The compiler will issue a warning, but the function gets executed. In such a situation, the parameters with the null values will almost certainly result in errors during the execution. The tool in this work catches these errors preemptively.

Fig. 1.2 shows an example of how function calls may be made using a variable number of arguments. Inside the function body itself, PHP supplies the methods: `func_get_args()`, `func_num_args()`, and `func_get_arg()` that can be used to work with the arguments supplied. The method, `func_get_args()` returns the function arguments list as an array, while the method `func_num_args` returns the number of arguments supplied. The method `func_get_arg()` takes an integer argument and returns the specified argument from the argument list. Since PHP 5.6, there are newer ways to handle and process variable arguments lists, but we will limit the discussion to the mentioned approach for
Functions are used extensively in the PHP language. In effect, the language itself was constructed on the principals of using functions. However, the dynamic nature of functions as discussed earlier in this section may introduce many potential errors in a PHP application. Consider the following definition in Fig. 1.3.

There is a potential error with the script in Fig. 1.3. If $some_condition in the body of function function_call does not evaluate to true, then the function call in line 12 does not yield any value. Consequently, $result will have the null type. Trying to access $result anywhere else in the script will cause the PHP application to almost certainly crash. In a worst case, this null value may be internally coerced to a false value or yield subsequent null values. When the application ultimately crashes or delivers a wrong result, the failure site will be a long way off from the bug.

A second way in which an error may be introduced is by calling a function as below:

```php
$result = function_call($some_value);
```
1.5. CLASSES AND OBJECTS

Classes and Objects form the basic building blocks for many modern programming languages. Together, they are a core set of constituents for the Object Oriented Programming (OOP) model that...
many modern programming languages are based on. PHP was meant for providing developers the tool to develop websites quickly and efficiently. Starting in version 4 of the language, basic class features were added to the language. Starting in PHP 5, support was added for a full fledged OOP environment including features such as visibility, abstract and final classes and methods, interfaces, and inheritance. At this moment of writing, the most recent PHP release is version 5.6.6 which came out on Feb. 19, 2015.

A PHP class is a collection of functions and associated variables. In that way, it is not different from classes in mainstream OOP languages such as Java and C++. Classes provide a way to introduce user-defined types in an application. An example of a simple class definition is given in Fig. 1.4.

Functions in classes are referred to as methods, and the associated variables are called properties. Class members (methods and properties) are associated with visibility modifiers that affect the way the members can be accessed and used. Additionally class members can optionally be specified to be of the static type.

The introduction of class types adds a whole new dimension to the kinds of errors that can result in a PHP application. Unlike values of primitive types, classes feature a set of properties, each one of which may be any primitive or a user-defined type. Classes may also inherit property and method definitions from a parent class. Thus, class types may introduce complex type-hierarchies and type interdependencies in the application. This makes static typing of class types particularly difficult. In the rest of this section, we provide more discussions on some of the ways in which using class types can result in type and other semantic errors in a PHP application.
<?php
// beginning of class definition
class Employee{
    // a public property
    public $name;

    // a private property
    private $empID;

    // getters and setters
    public setName($val){
        $this->name=$val;
    }

    public setEmpID($val){
        $this->empID=$val;
    }

    public getName(){
        return $this->name
    }

    public getEmpID(){
        return $this->empID;
    }
}
?>

Figure 1.4 Example of a class definition in PHP
1.5.1 Static and instance class members

Static class members in PHP are quite similar to their counterparts in mainstream object-oriented languages (Java, C++ etc). The values in static properties are shared by all instances of a class. Public static properties can be accessed outside the class by prefixing their names with the particular class that includes them. Both private and public static properties can be accessed within the class definition by prefixing their names with the keyword `self`. Similarly, using the keyword `parent` inside a class definition lets an expression access any static property defined in its immediate parent class.

Static methods are also accessed the same way as static class properties. Since static methods are not associated with any object instance, static method bodies may not contain any reference to the `this` identifier. PHP issues a run-time error if it encounters the `$this` expression in a static method. An illustration is given in Fig. 1.5.

```php
class A{
    public static $static_var;
    public $obj_var;

    public static function modify_var($val) {
        self::$static_var = (self::$static_var - val)+100;
    }

    public static function modify_obj_var() {
        // error: static method is not associated
        // with any specific object instance.
        return $this->obj_var;
    }
}
```

Figure 1.5 Examples of using static class members in PHP
Consider the method: `modify_obj_var` in Fig. 1.5. It is a static function, but includes a reference to a `$this` identifier in its body. If the PHP compiler executes this statement, it throws a fatal runtime error and crashes. A script fragment that may induce such a crash, despite seeming innocuous, is as follows:

```php
$obj = new A();
$obj->modify_obj_var();
```

Instance class properties, in the meantime, are associated with specific class instances (or objects). Values in instance properties are not shared among the objects created from a class. Each object will maintain its own copy of all the declared instance properties. Inside class methods that are not declared static (or instance methods, in short), each object may access its instance property by using the `$this` identifier. Instance methods may also reference static properties using the `self` identifier, or the name of the class that owns the static property.

There are many ways in which problems may arise with using static and instance members of a class type. For example, any property or method that is misspelled may cause fatal runtime errors. Or, it maybe that the static and instance members are not used in the right context. It may also be that certain expressions violate the visibility restrictions placed on class properties. Strongly typed languages such as Java and C# will usually catch such errors during compile-time. In part, our tool is also designed to exhibit such characteristics. While it is impossible to offer the same kind of rigorous type-validations as in statically typed languages, our tool attempts to do the best it can to report possible erroneous sites.

### 1.5.2 Object Aliasing

As of PHP 5, object variables no longer actually contain the object that are assigned to them in a program statement. Instead, such program variables contain a copy of an object identifier resource (or object accessor) which lets them access the actual object. Likewise, when an object variable is copied into another program variable, sent as a function call argument, or returned from a function body, it is not the actual object, but the object accessor that is copied, passed, or returned in such an expression [Obj].

As a result of the above, two or more program variables in PHP could alias the same internal object.
In this situation, the object state could be changed using any one of the aliasing program variable. Any subsequence object reference must reflect the updated value of the object. This is shown in Fig. 1.6.

```php
<?php
//define a class
class Test {
    //a class property
    public $var
}

//create an object
$a=new Test();
$a->var=99;

//create alias
$b=$a;
//modify object state using alias
$b->var="100";

//Initial object reference
//show updated object state
echo $a->var; //print "100";
?>
```

**Figure 1.6** Object referencing in PHP

Using object aliases, and particularly misusing them, has the potential to create extremely hard-to-trace bugs. While the type analyzer in this tool adequately tracks such aliases through a script, the subject of correctly using them (or misusing them) is largely an open-ended matter. No errors are flagged just from having aliases in the program.

However, we do believe that there is one pattern involving aliases that is error-prone, and deployed features to flag them during type analysis. When object variables are passed as function arguments, any changes made to the object state in the function body persist beyond the function call. Specifically, when such cases feature changing the type any of its (passed object’s) properties, then it is
likely that this may result in an error later in the script.

### 1.5.3 Other sources of errors

Classes and objects may be the sources for many other kinds of hard-to-trace errors in a PHP application. The dynamic nature of PHP allows for some interesting object constructs. For example, new instance properties can be dynamically associated with an object. Consider the following,

```php
$a->second_var = "some_content";
```

The statement above will associate a new property by the name, `$second_var` to the object referenced by `$a`, and by `$a` only. A different object created from the same class will not have such a property. Even worse, if the second object tries to access this property (and in general, any property that is not defined), the PHP compiler will silently yield a null value in such a case and continue execution. We believe that both these features have the potential to introduce errors in the application, and developed features to detect such cases.

Another potential error that may result from the dynamic nature of PHP is a construct where methods and properties may be invoked on a non-object program variable. Consider the situation in Fig. 1.7.

```php
<?php
...
if ($cond) {
    $varA=new A();
} else {
    $varA="some_string.";
}

// try to invoke method on $varA
$varA->modify_var(99);
```

**Figure 1.7** Calling a method invocation on a non-object variable.
In this chapter, we reviewed many vulnerable features of the PHP language. This list is not intended to be exhaustive, for the PHP language has evolved significantly since its inception in 1994 and supports a large variety of program constructs at the moment of this writing. PHP is a dynamically typed language ideal for server-side application development. Its features make it relatively easy for website developers to get a web application up and running. There is minimal overhead in writing and maintaining boilerplate code, so that developers can focus on application logic and UI contents. The most modern versions of PHP support the full object oriented programming model, and this has been the basis for many large scale PHP applications and frameworks. While we sum up our review of the PHP language in this section, we will explore the features of a new programming language in the next section. This language is called 'Rascal’ and it serves as the primary analysis tool for this work.
CHAPTER 2

RASCAL

2.1 Introduction

The Rascal [Kli09] programming language is an experimental general purpose language that is currently being developed and enhanced at CWI in The Netherlands. While its design is based on many earlier languages, Rascal is a direct successor to the ASF+SDF specification language [Bra01].

The language features many core elements of traditional imperative languages such as variables, functions, and scoping rules. This makes it quite easy for any developer to get started on the language. The language provides many advanced language processing utilities such as context-free grammar definitions, a built-in parser generator and syntax tree generator that makes it especially useful for meta-programming tasks such as language analysis [HK14], code re-factoring [Hill12] and creating domain-specific languages [Hill14a]. The language features analysis tools based on relational calculus, relational algebra, and logic programming systems such as Crocopat [Bey06], Grok [GM09], and RScript [Kli08] out of the box. These, with support for modern programming
features such as polymorphic functions, generics, late binding, and pattern-matching make Rascal a powerful meta-programming language of choice.

In the next few sections, we provide discussions aimed at introducing some of the core features of the language. The interested readers are advised to refer to the official documentation for more information [Rscd].

2.2 Built-In Type System

Rascal's type system is based on immutable statically typed variables [Goe06] whose values never change. The data types are derived from a custom type lattice. This lattice is presented in Fig. 2.1. At the bottom of this lattice (shown on the left) is the type `void`, and at the top (shown on the right) is the type `value`. These data types follow a hierarchical relationship where the direction of the arrows in the type lattice graph denotes a `subtype-of` relationship. For instance, `real` and `int` types are both sub-types of the `num` type. To put this in reverse, the type `num` is a generalization of the types `int` and `real`.

From Fig. 2.1, we see that Rascal provides a rich set of built-in data types. The primitive types from mainstream languages are provided by the `bool`, `int`, `float`, `num`, and `str` types. Collection types are provided by the `list`, `set`, `tuple`, and `map` types. More complex types such as `relation`, `nodes`, and `algebraic data types` are also available. We will provide some discussions on some of the interesting Rascal primitives in the remainder of this section.

2.2.1 The `loc` type

In particular, the `loc` type is an important Rascal primitive. It provides a consistent way to access contents in local and remote files. This data type consists of two fragments: the URI of the file, and optional values that locate a specified range of characters in the file. The syntax for a `loc` type is given as follows:

\[
loc a = |\text{uri}|(O, L, <BL, BC>, <EL, EC>)
\]
Figure 2.1 Rascal type system lattice [Rscb].
where,

- \textit{uri} is the Uniform Resource Identifier expression,
- \(O\) and \(L\) are integer expressions giving the offset of this location from the beginning of the file and its length, respectively,
- \(BL\), and \(BC\) are integer expressions giving the beginning line and column of the location, and
- \(EL\), and \(EC\) are integer expressions giving the ending line and column of the location.

This type is quite significant because it allows one to associate extracted facts with exact code locations in a source file. In this work, the AST of the target PHP code is available as a Rascal data structure. Each node in that AST has a \textit{loc} data value that precisely maps to the text in the corresponding source file. This information could be used to annotate the specific text to suggest if an error is detected in that content. Although such live text annotations are not already built into the tool, it could be achieved with a small effort.

2.2.2 Rascal collection primitives

The \textit{list} type in Rascal allows the storage of an ordered sequence of uniform type elements. A \textit{list} may contain more than one instance of the same element. A Rascal \textit{set} is an un-ordered collection of unique elements. Like a list, a set may also contain elements of only homogeneous data types. The syntax for lists and sets are given in 2.1 and 2.2 respectively.

\[
\text{A Rascal list} : [\ Exp_1, \ Exp_2, \ldots, \ Exp_N \ ] \tag{2.1}
\]

\[
\text{A Rascal set} : \{ \ Exp_1, \ Exp_2, \ldots, \ Exp_N \} \tag{2.2}
\]

where,

\[
\text{typeof}(\ Exp_1) = \text{typeof}(\ Exp_2) = \cdots = \text{typeof}(\ Exp_N).
\]

The \textit{tuple} type in Rascal has no counterpart in traditional languages such as Java and C++. A tuple is an ordered collection of elements. Unlike the elements of a list or a set, the elements of a tuple can
be of different types. Each element may also have a label that can be used to access that element
directly from the tuple that it belongs to. The syntax for a tuple in Rascal is given in 2.3.

\(\langle \text{Exp}_1, \text{Exp}_2, \ldots, \text{Exp}_N \rangle\) \hspace{1cm} (2.3)

A relation type in Rascal is a set of tuples that have the same static type. Since they inherit from the
set type, they support all operations and functions applicable to sets. However, they also expose
additional functions that the user may use for solving relational problems. A relation type may be
instantiated by declaring a set of similar tuples, or through some operations: for e.g. taking the
cartesian product of two sets. The syntax for a relation in Rascal is given in 2.4.

\(\{\langle \text{Exp}_{1,1}, \text{Exp}_{1,2}, \ldots, \text{Exp}_{1,N}\rangle, \langle \text{Exp}_{2,1}, \text{Exp}_{2,2}, \ldots, \text{Exp}_{2,N}\rangle, \ldots, \langle \text{Exp}_{M,1}, \text{Exp}_{M,2}, \ldots, \text{Exp}_{MN}\rangle\}\) \hspace{1cm} (2.4)

where,
\(\text{typeof}(\text{Exp}_{1,i}) = \text{typeof}(\text{Exp}_{2,i}) = \cdots = \text{typeof}(\text{Exp}_{M,i})\) for \(i = 1, \ldots, N\).

A map type in Rascal is a set of key:value pairs and is functionally equivalent to that in traditional
languages like Java and C++. The keys and values can be of different static data types. This is typically
not allowed in languages such as Java and C++, where by language requirements, a constant static
type for keys and for values are declared beforehand.

It is also possible to declare a map variable in Rascal with expected static types for the keys and
values. In this case, the elements in the set of keys and values each have to have the type referenced
in the map declaration. The syntax for a map type is given in 2.5.

\( (\text{key}_1 : \text{value}_1, \text{key}_2 : \text{value}_2, \ldots, \text{key}_N : \text{value}_N ) \) \hspace{1cm} (2.5)

### 2.2.3 Algebraic Data Types (ADTs)

Rascal ADTs are certainly one of the most important Rascal types used in this work. Critical analysis
components, including but not limited to, the control flow graph (CFGs) and ASTs of the target
PHP code, the systems of type-based constraints, and the modeled concrete types themselves, are
2.2. BUILT-IN TYPE SYSTEM

defined using Rascal ADTs. They allow users to define problem specific data types through the use of polymorphic data constructs. The syntax for ADTs in Rascal is given below,

\[
data Name = alt_1 | alt_2 | \ldots | alt_n;
\]

(2.6)

where

\[
alt_i = name_i (type_1 param_1 | type_2 param_2 | \ldots | type_N param_N)
\]

(2.7)

The declaration in 2.6 defines a new algebraic type called \( Name \). This type can now be used like a static type for any program elements such as variables, lists, sets, tuples, return types for functions, and even parameter types for other constructors in the program. A new instance of this type can be created by using any one of the constructors defined for this type. These kinds of functions are called the constructor functions and follows the syntax as listed in 2.7.

A constructor function can take any number of statically typed parameters. The parameters act as storage mechanisms for the new instance of this ADT and their values can be accessed by using the dot operator. For example in 2.8 if \( a \) is an instance of an ADT created with a constructor function that had \( param_i \) as a parameter, then the following expression can be used to get the value in \( param_i \),

\[
Type_i b = a.param_i.
\]

(2.8)

Functionally, the ADT: \( Name \), with its set of constructors (2.6), can be thought of as a Java system where a parent class by the name \( Name \) is sub-classed by a set of other classes, specifically, classes with names: \( alt_1, \ldots, alt_N \) in this case. The constructor function in 2.7 can be viewed as a public constructor provided for a corresponding sub-class in the considered Java system. In this view, the operation of creating a new instance of ADT: \( Name \) is equivalent to creating a new instance of on of its sub-classes. This instance can then be passed to any Rascal expression that takes data of the \( Name \) type.
2.3 Program Constituents

Rascal scripts are written in modules. A module is a file containing program source code in Rascal. A Rascal module may contain any valid Rascal declaration or statement. Modules can be organized by packages as in Java. Figure (2.2) gives all the top-level statements that can be included in a module.

Figure 2.2 Top-level Statements in Rascal Modules [Rsca].
2.3. PROGRAM CONSTITUENTS

2.3.1 Import

Rascal modules can import other modules to reuse functions, data types and any other declarations made available in the module being imported. Rascal also supports local and global scoping. A function or a data type is made public by default in Rascal, and this content is available to any module that imports the module. A module needs to explicitly import each module that it intends to use. Transitive module imports are not supported in Rascal.

```
import QualifiedName;
```

Figure 2.3 Syntax for Import Statement in Rascal

2.3.2 Syntax definition

A particular application of the Rascal language is also in the creation of domain-specific languages, or DSLs. A DSL [Fow10] is a special-purpose language meant to solve a domain-specific set of problems. Rascal can be used to define the grammars of such a DSL using the syntax definition construct. This discussion is beyond the scope of the current work, and we direct the interested reader to the official Rascal documentation for more information on this topic [Rsc].

2.3.3 Variable

A Rascal variable is very similar to variables in other mainstream languages. A variable declaration has the syntax as listed in 2.9.

```
Type Name = (Exp);
```

(2.9)

A variable has the scope to the nearest enclosing block. If the variable appears as a top-level statement in a module, it is available everywhere in the module, including inside functions. Re-declaration of a variable is not allowed in the same scope. The variable will also be available in all modules...
that import the module it is declared in, as long as its declaration includes the public keyword modifier.

```rascal
//formal variable declaration in Rascal
int aVar=10;

//variable definition without static type declared
bVar= "string_value"; //assign a string value to variable: aVar
....

bVar=1.24; //ok!
aVar=false; //error! aVar is declared to be of the int type
```

**Figure 2.4** Variable Typing in Rascal

### 2.3.4 Function

A function in Rascal is a top-level module statement and its syntax is listed in Fig. 2.5. In many regards, a Rascal function is akin to functions in traditional language. Function definitions have a set of statically typed arguments and a return type. Function definitions can also be overloaded.

```plaintext
modifiers Type Name (list [parameter]?) (throws list [exception])? {body},
body := statement | expression | list [body]
modifiers := ("public" | "private")? ("java" | "test" | "default")?
parameter := Type Var | Pattern
```

**Figure 2.5** Syntax for Functions in Rascal

If more than one type of data is to be returned from a function, it can be done so by returning a tuple (section 2.2.2). Each element of a tuple can be retrieved by either projecting the tuple on its index,
or using the name of each of its fields. This offers a simple solution to the problem of returning multiple values from a function. An example is given in Fig. 2.6.

The optional keyword `java` declares that the body of the function is implemented in Java. The keyword `test` specifies that the given method is a test function with the requirement of a boolean return type. Rascal provides a built-in automated testing framework which enables developers to execute all test methods in scope by running the `:test` command in the console. Finally, the optional keyword `default` signifies that the provided function definition should be used when all other overloaded function definitions fail to match a call signature. This is relevant when the function parameters consist of patterns instead of formal type arguments. The discussion of patterns as function arguments will be deferred to section 2.4, where we formally treat patterns.

```java
// definition of a function that returns a tuple.
public tuple[int rank, str myValue] getRankAndValue(int aVar)
{
    ...
    return <someRankValue, someStringValue>;
}

// call function
returnedValue = getRankAndValue(someIntegerArgument);

// retrieve tuple values using field name
gotRank = returnedValue.rank;
gotValue = returnedValue.myValue;

// or, one may directly write
<gotRank, gotValue> = getRankAndValue(someIntegerArgument);
```

**Figure 2.6** Definition of a Function Returning a Tuple Type

In this section, we briefly reviewed the anatomy of functions in Rascal. Functions play an indispensable role in Rascal for writing modular code. They allow many customization that make them quite useful. Among others, for example it is possible to return an arbitrary number of data type from a Rascal function by returning them as elements of a tuple. It is also possible to import function
definitions written in plain Java, and Rascal also makes writing automated tests simple and efficient. Furthermore, by using patterns and algebraic data types as function arguments, one can write polymorphic function definitions.

### 2.3.5 Alias

The alias keyword is used in Rascal to give a meaningful name a certain type or a combination of types. The purpose of using an alias is completely for better readability of the code. The syntax for alias is given in (2.10). A very simple usage example is given in Fig. 2.7.

\[
\text{alias Name} = \text{Type;}
\]  

(2.10)

```
//define alias for the int type in the script
alias StudentID = int;

//declare two int variables using the alias
StudentID id1 = 100;
StudentID id2 = 200;
```

**Figure 2.7** Using Alias to Define New Reference for a Type in Rascal

### 2.3.6 Annotation

An annotation is a piece of information that can be attached to an exiting node value. The utility of this is in dealing with data structures defined externally and whose redefinition could potentially break other parts of the code that depend on these data types. Thus, annotations provide a minimally invasive approach to solve the problem of giving nodes additional information. The syntax for an annotation is as follows,

\[
\text{anno annoType OnType@name}
\]
2.4 Patterns and Pattern-Matching

Pattern matching is an important Rascal expression that evaluates to a boolean value. This expression can match a variable to a number of different kinds of value, and in the event that the match succeeds, required actions can be taken. Such expressions have been widely used in this work, ranging from matching variables to specific constructor types in an ADT to deep nested searches on generated ASTs.

The use cases of patterns in Rascal can be very broadly divided into their usages as control expressions and function arguments. A brief discussion of each is provided in the next two sections.
2.4.1 Patterns as control expressions

Patterns provide users a way to match variables to any literals, collections, or a user-defined data type in the program. In addition to evaluating a boolean expression based on the pattern match, pattern matching bind variables to specific data elements in the pattern. Examples of these concepts are illustrated in Fig. 2.9. In Fig. 2.1, we present the general and specific pattern matching syntax for all available patterns in Rascal.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>Bool, Int, Real, Num, String, Loc, DateTime</td>
</tr>
<tr>
<td>Variable declaration</td>
<td>Type Var</td>
</tr>
<tr>
<td>Multi-variable</td>
<td>*Var, *Type Var</td>
</tr>
<tr>
<td>Variable</td>
<td>Var</td>
</tr>
<tr>
<td>List</td>
<td>[ Patt₁, Patt₂, ..., Patt₃ ]</td>
</tr>
<tr>
<td>Set</td>
<td>{ Patt₁, Patt₂, ..., Patt₃ }</td>
</tr>
<tr>
<td>Tuple</td>
<td>&lt;Patt₁, Patt₂, ..., Patt₃ &gt;</td>
</tr>
<tr>
<td>Node</td>
<td>Name ( Patt₁, Patt₂, ..., Patt₃ )</td>
</tr>
<tr>
<td>Descendent</td>
<td>/ Patt</td>
</tr>
<tr>
<td>Labelled</td>
<td>Var : Patt</td>
</tr>
<tr>
<td>TypeLabelled</td>
<td>Type Var : Patt</td>
</tr>
<tr>
<td>TypeConstrainted</td>
<td>[ Type ] Patt</td>
</tr>
</tbody>
</table>

The general syntax for using abstract pattern given in 2.11. The general rule is that the desired pattern is defined on the left-hand side, while the interesting variable is placed on the right-hand side of the pattern match expression. If the match succeeds, then the expression evaluates to true and any block statement following will be executed. Fig. 2.9 gives some examples of using abstract patterns

\[(Abstract\ Pattern) :=\ \text{variable}\ \]  \hspace{1cm} (2.11)

Fig. 2.9 illustrates several key elements. First, in lines 9 and 12, pattern matches are used to evaluate if the argument passed to the function, \textit{elem}, is one of the specific constructor types used in the definition of ADT \textit{Graph}. Secondly, also used in these pattern match expressions are the bindings of
2.4. PATTERNS AND PATTERN-MATCHING

```java
public data Graph = Node (int id) | Edge (int from, int to) | ... ;

Graph aNode=Node(1) ;
Graph anEdge = Edge(1,2);

public void function printGraphElement(Graph elem){
    if ( Node(int myID) := elem )
        println("Node_ID:<myID>.");

    else if ( Edge (int fromID, _ ) := elem )
        println("Edge_from:<fromID>.");

    else
        throw "Unsupported.";
}

printGraphElement(aNode);  // prints: Node_ID:1
printGraphElement(anEdge);  // prints: Edge_from:1
```

Figure 2.9 Pattern matching usages in Rascal
locally scoped variables: myID and fromID in lines 9 and 12 respectively. These variables only have scope within the block defined for the match condition. When one of these patterns evaluate to a true, then value of the variable (myID or fromID) assumes the value of the constructor parameter of the data stored in elem. Lines 19 and 21 illustrate how this works.

Lastly, it can be seen from Fig. 2.9 that one can use the underscore (‘_’) character to ignore one or more of any of the constructor parameter in a pattern match expression. In line 12 of Fig. 2.9, the underscore character instructs the Rascal interpreter to ignore the specific data that maybe stored as the second parameter of elem. The underscore character is still required in that position to signify the presence of a parameter. Without the underscore character, the pattern would fail to match as intended.

2.4.2 Patterns as function parameters

Earlier, we mentioned that patterns may also be used as parameters in function definitions. The parameter may consist of any of the abstract pattern declared in Table 2.1. This usually leads to the definition of a set of related polymorphic functions. What is special about using such a construct is that one can define a default function definition which will be called the call arguments do not match any pre-defined patterns. Some examples of set of related pattern-based function definitions are given in Fig. 2.10 and Fig. 2.11.

![Figure 2.10 Example of using patterns as formal function parameters](image-url)

```rascal
data Exp = Foo(int val) | Bar(int chr)
         | Add(Exp E1, Exp E2) | Exception();

//Define a set of polymorphic functions based
//on formal pattern parameters

//formal parameter matched to a 'Foo' constructor
Exp compileExpression( Foo( int val ) ) = return Foo( val );

//more pattern-based definitions...
Exp compileExpression( Bar( str chr ) ) =
    return Foo(((100%chr + 32) - 1.2));
```

Figure 2.10 Example of using patterns as formal function parameters
There are a couple of other ways in which patterns may also be used in Rascal. Notably, *concrete pattern matching* is a particular use case involving the parsing of a program text written in a different language. In order to do this, Rascal needs to know about the external program's features, supplied through its syntax definition and non-terminal declarations. Another use cases of patterns is manipulating strings using *regular expressions*. In summary, pattern matching is a powerful Rascal feature that can be used to simplify a lot of tasks. It is also a very versatile mechanism and can be used in many different ways to write short expressive code.

```plaintext
Exp compileExpression ( Add( Foo( int val1 ), Foo( int val2 ) ) ) = return Foo ( val1 + val 2);

Exp compileExpression ( Add( Foo( int val1 ), Bar( int chr2 ) ) ) =
  return Foo ( val1 + compileExpression(chr2));

Exp compileExpression ( Add ( Bar( int chr1 ), Foo( int val2 ) ) ) =
  return Foo ( compileExpression(chr1)+ val2));

Exp compileExpression ( Add ( Bar( int chr1 ), Bar( int chr2 ) ) ) =
  return compileExpression
  (compileExpression(chr1) + compileExpression(chr2) );

//define a default definition.
default Exp compileExpression ( Exp e ) = return Exception();
```

*Figure 2.11* More pattern-based function definitions from Fig. 2.10.

### 2.5 Conclusion

In this chapter, we reviewed the Rascal programming language and some of its main features. This language provides a powerful platform for performing meta-programming tasks such as language analysis and source refactoring. It features a number of advanced built-in utilities such as a parser and syntax tree generator that frees users from having to write their own language processors. Additionally, Rascal provides a rich set of built in types and functions to aid in complex analysis tasks. That, along with the availability of the PHPAir framework [HK14] makes Rascal very well suited
for our type analysis task for PHP.
3.1 Analysis of Methodology

3.1.1 Type Systems

Type systems [Pie02] allow for the proper type inference of expressions and values in a programming language. They are required to check against incompatible types that may appear in an expression, e.g. string data used in an arithmetic operation. When such a case is encountered, the underlying type system is responsible for deciding what actions to take. For example, it may flag an error to let the user know, or it may use an applicable type coercion rule to proceed with the evaluation of the expression. This kind of type checking can happen during different moments of the life-cycle of a program. Depending on this, type systems can broadly be divided into two types,

- Static type systems: This kind of type system involves generating types at compile-time, wherein the types of all the expressions have been deduced before the program is run. Having this knowledge not only allows the compiler to give type feedback to the developers on any er-
roneous constructs, but also to generate fast and efficient compiled codes. The disadvantages
of such a system are usually in losses of code expressivity and incurred computational effort
for tasks such as type-inference, compilation, and linking.

Statically typed systems are usually based on variations of the Hindley-Milner algorithm
[Mil78] for type inference. It was first implemented for the functional language ML. This
algorithm was developed for a language where the types of expressions are not explicitly
declared. This is in contrast to popular languages like Java, C and C++ where the types of
expressions are declared at compile-time. In the case of the latter, no type inference algorithm
is particularly required. In either cases, the fact that type information is known at compile-
time renders powerful type-checking and optimization capabilities to the language compiler.

For applications where execution speed is critical, or there is a need to maintain large code
repositories, statically typed languages are often chosen over dynamically typed languages.

- Dynamic type systems: Dynamic type systems work by deducing the type information of
expressions during run-time. In this case, run-time checks are performed at each expression,
and should such a check fail at some location, the compiler will likely terminate execution.
This may result in huge loss of computational time and data, and begets the need for type
checking systems such as is developed in this work.

In the meantime, dynamically typed language allow for more expressive and highly poly-
morphic code. Because of the dynamic nature of these languages, programs also tend to be
shorter, and consequently, more manageable. The risks they incur at the expense of these
amenities, however, are codes that may be more likely to crash during runtime, and codes that
are usually slower than those in statically typed languages. Some examples of programming
languages featuring a dynamic type system are Python, Perl, Javascript, and of course, PHP.

The type system featured in this work is more aligned with a type system that takes inspirations
from both the systems described above. This system, referred to in literature as the soft typing system
[Wri96; CF91; Aik94] is designed to offer the benefit of static typing to dynamically typed languages.
This is accomplished by using a custom type inference algorithm to figure out as much as is possible
statically, and insert run-time checks for the rest. This resolves both the ills with dynamic languages,
i.e. the issues of them being slow and prone to runtime failures, while preserving the factors that
make dynamically typed systems desirable.
3.1.2 Related Work

In [Wri96], a modified version of the Scheme compiler is built using the principles of soft typing systems. Soft Scheme, as the author calls it, provides global type checking for Scheme applications, thus providing explicit type-checks and eliminating about 90% of run-time checks. They report that this lead applications to run up to 3.3x faster than ordinary dynamically typed applications.

Other works have taken a slightly different approach to speed up the performance for dynamically typed applications. This class of approaches actually involves transforming the source application to an equivalent application on an entirely different platform. The target platform is usually chosen to be a fast statically typed system such as C++ or Java. In addition to generating semantically equivalent code, this also involves generating concrete types for any identifiers, functions, and methods in the dynamically typed source language, to compatible types in the statically typed target language. Starkiller, a python-to-C++ compiler [Sal04], is an example of such a source-to-source compiler. In 2010, Facebook [Fac], the social network behemoth, revealed the Hip-Hop compiler for PHP-to-C++ transformation [Hip]. Just three years later, Facebook retired the Hip-Hop compiler to feature a new JIT engine for PHP: HHVM (an acronym for the HipHop Virtual Machine) [Hip]. The social giant claims that this platform provides upto a 9x speed performance for PHP applications.

Ecstatic [MMK07] is an example of type inference tool developed for the Ruby language. The tool aims to derive types for Ruby using the cartesian product algorithm (CPA) [Age95; Age94]. This is mostly possible because of the close similarities between the 'Self' language [Age93], which the CPA algorithm was originally designed for, and Ruby. They’re both strongly object-oriented languages where types may be modeled as the list of messages that an object understands [BI82]. The authors of Ecstatic suggest that this approach provides a more precise type model for Ruby objects than, say, assigning parent classes as the object types. This is premised on the fact that the dynamic features of Ruby allow objects to be modified to receive specific additional messages, thus, adding to the kinds of messages that a Ruby object may understand during runtime.

Another dynamic language that has spurred a number of static type checkers is the Javascript language. It is a highly dynamic language with support for prototype-based object-oriented features [Ric10], as in Self. It is one of the most popular language in the programming community. In fact, Github currently lists the largest number of active repositories for Javascript[Git], including the language with the fastest growth rate of online repositories in the last two years. Javascript serves a
unique niche in the programming multiverse as a one-of-its-kind platform for generating dynamic web-content.

A quick search on Google using the keyword "type inference javascript" also results in a large list of static type-checker prototypes for Javascript. Facebook's flow [Flo] and Mozilla's SpiderMonkey [HG12] seem to be some of the more prominent ones in the list. While flow is designed with the same kind of utilities as the tool in this project, i.e. provide type-feedback on user-generated scripts, spider-monkey is the javascript engine in Firefox, Mozilla's flagship web-browser, and uses its type-checking capabilities to perform internal run-time optimizations.

The list of works on type-inference for PHP, in the meantime, is comparatively much more limited. Facebook's HHVM is essentially a compiler for Hack, an extension to PHP Phantm [Kne10] is another work we found in the literature offering some level of static-type inference support for PHP. Perusing through the reference led us to infer little aside from that the tool specifically checks for runtime type-mismatches. We believe that the tool developed in this work aims to offer a broader spectrum of runtime type-analysis than in Phantm.

The direction for this work is primarily based in the methodology developed in [Cam09]. In [Cam09], capabilities for several types of analyses for PHP applications are developed. They include tasks such as HTML validation, enforcing of proper coding practices and protocols, and an implementation of a static type inference algorithm. The former two analyses largely involve string-based pattern searches using regular expressions. This approach allows for a fast, high-level processing capability that includes, but is not limited to, tasks such validating that all HTML tags are properly closed, or that only tags confirming to the W3C standards are used, and more complex scenarios such as verifying that minimum standards are used for verifying and validating input data, cookies, database, file resources, and so on.

The analysis involving static type inference is a much more involved one. For that purpose, [Cam09] resorts to an algorithm based on using constraints and a set of concrete types to generate and model types from PHP expressions. We follow suit, but our implementation is developed from scratch, and possibly varies from the referenced work in significant ways. For instance, our treatment of function calls is quite different. More notably, we've extended the type language and the constraints system to be able to accurately model class-types in PHP. In the rest of this chapter, we give details of the methodology of our work.
3.1.3 Type Language

Our analysis is based on a type inference language that treats values as having a set of concrete types in the analysis. A concrete type is the most specific type that can be assigned to a value. The foundational set of such concrete types contain the primitive data types from C-like languages, i.e. *strings*, *booleans*, *integers*, *float*, and *arrays*, and user-defined types in the form of *Classes*.

By keeping track of the set of concrete types, this type language is capable of more accurately determining what the possible type of any variable or expression may be in the PHP program. This is not possible in PHP’s way of storing information where values are stored as strings and hashmaps. The possible types of a program variable or expression are modeled as sets because PHP allows program constructs where it is possible to assign different types of values to the same variable. The same is possible for the return value of a function where values of different types may be returned from the function body.

A precise concrete type cannot always be evaluated for PHP expressions without having all the run-time information. The best we can do, at these situations, is to collect all the possible types as elements of a type-set. Consequently, it is understood that the value of the variable or the expression could be one of either of the types in the set. Some examples of this are given in listing 3.1.

In our type language, the base set of concrete types are given by the set of elements listed in Fig. 3.2. These elements are represented by unique constructor functions defined for the algebraic data type: *Type*. The elements: *Bool*, *Int*, *Float*, *String*, and *Array* are intended to have a one-to-one mapping with the primitive types: int, float, string, and array encountered with most imperative C-like languages, respectively. The element *Num* is intended to be a general type for both *Float*, and *Int*, and is included for representing a type that could either be a float, or an int.

The element *Object* is intended to map to a value that is an *object* in OOP languages. It is to be noted that while an *object* is not necessarily a type, it also does not suffice to generalize an object value as simply having the type of its class. This is because two objects of the same class could be structurally different, owing to the possibility of their fields having different types. Consequently, we assign unique objects as types for variables and expressions that may yield them. The parameter *h* serves to uniquely identify an object type, and its details will be discussed in a later section when we delve into the *Object* type.
3.1. ANALYSIS OF METHODOLOGY

CHAPTER 3. TYPE ANALYSIS FOR PHP

```php
<?php
function getValue($param)
{
    if ($param)
        $b = "myString.";
    else
        $b = 99;

    // Value of $b could be either a string, or an integer.
    return $b;
}

// $val could be either a string, or an integer!
$val = getValue($arg);
?>
```

Figure 3.1 Example of a case when a PHP variable may have more than one type

```php
public data Type =
    Bool() | Int() | Float() | String() | Num() |
    Array(Type t) | Object(Handle h) | Any() | Void();
```

Figure 3.2 Base set of concrete types in the Analysis
3.1. ANALYSIS OF METODOLOY

The type *Any* is used to denote a value whose type could be any of the concrete types. In our analysis, a PHP program variable can assume this type in a couple of ways. This type may result from a sequence of type unification processes (discussed later), from a recursive function call sequence, and any case when the type checker is unable to determine a concrete type-set for an expression. Finally, the type *Void* is applied to any expression that results in a *null* PHP value. This could happen because of a number of reasons, for e.g. by assigning a value from a function that does not return any value, or from a variable that has not been initialized in the script.

3.1.4 Method Overview

The analysis in this work, in the simplest view, can be thought of as a fixed point algorithm that iterates until the types do not change for any of the program points. The fixed point iteration (FPI) method can be mathematically expressed as:

$$\Psi^{n+1} = \Gamma \Psi^n$$  \hspace{1cm} (3.1)

where $\Psi^{n+1}$ and $\Psi^n$ are the sets of types for all the program points in the $(n + 1)^{th}$ and $n^{th}$ analysis iterations respectively. $\Gamma$ represents the transformation that each set of types undergoes in one such iteration and can be thought of as a matrix operator. In the beginning each program point is assigned an empty set of types. During type iterations, types are be added to these sets. The process of adding types to sets is *monotonic* and so these sets can grow arbitrarily large.

In its basic form, this method of simple iterations is fundamentally flawed. Not only does this method create large unwieldy sets to work with, but it could also lead to non-terminating analysis iterations. A short example of this is presented in Fig. 3.3. In this example, the type analysis would iterate indefinitely while trying to infer a concrete return type of variable $a$ inside the *while* loop. The reason is very simple: each iteration would return an *Array* type of the previously inferred type for this point. The analysis would attempt to return an infinitely nested Array type, i.e. *Array (Array (Array (…)))*, thus leading to a non-terminating sequence of iterations.
3.1.5 Type unification

The problem of non-terminating iterations with simple FPI (Fig. 3.3) can be remedied by using a simple, yet effective procedure based on type unifications. Type unification is the process of unifying independent types into a more generalized type representation. This generalization must be a correct approximation of the actual type set, prevent non-terminating iterations, and still provide some useful type information about a program point. This is achieved by using the widening operator [FNH05], and is defined in Fig. 3.5. The partial lattice representation of the types that results by using the given type unification procedure for \( k = 4 \) is given in Figure 3.4 [Cam09].

In Fig. 3.5, \( \psi_1 \) and \( \psi_2 \) are type sets for a program point that we need to unify, while \( k \) is a parameter that defines the maximum number of individual types that may appear in the resulting type set. Situations to merge two different type-sets may occur at the end of conditionally branched statements, looping constructs, and when inferring return types from function bodies. More generally, the type unification procedure is used whenever we need to merge type information of a program point resulting from two or more sources.

Some examples of applying type unification using \( k=4 \) are given in Figure 3.6.

In our analysis, as in [Cam09], we have used \( k=4 \). Also, by *an array depth of at least \( i \)*, we mean that all the types in the set must of the array type with at least \( i^{th} \) level of array nesting. Some examples of this are illustrated in Fig. 3.7.

The particular utility of using the unification operator is that information on merged types is stored with a user-defined level of precision. The alternate to that would have been to say that we get *any* type immediately, or carry on storing all the types without any unification. The former would converge faster, but would be less precise. The latter would result in cases with infinite iterations.
3.1. ANALYSIS OF METHODOLOGY

CHAPTER 3. TYPE ANALYSIS FOR PHP

Figure 3.4 Type lattice resulting from using the $\nabla_4$ operator.

Figure 3.5 Definition of the widening ($\nabla$) operator for an integer value $k$.

$$
\psi_1 \nabla_k \psi_2 = \begin{cases} 
\bot & \text{if } \psi_1 = \psi_2 = \bot, \\
\psi_1 \cup \psi_2 & \text{if } |\psi_1| + |\psi_2| \leq k, \\
\text{array}[\text{any}] & \text{if } |\psi_1| + |\psi_2| > k, \text{ and,} \\
\psi_1 \cup \psi_2 \text{ has only types with an array depth of at least } i \epsilon \{0 \ldots n\} 
\end{cases}
$$
3.1. ANALYSIS OF METHODOLOGY

### CHAPTER 3. TYPE ANALYSIS FOR PHP

\[
\{\text{int}\} \triangledown_4 \{\text{float}\} = \{\text{num}\}
\]
\[
\{\text{int}, \text{bool}, \text{string}, \text{array}[\text{int}]\} \triangledown_4 \{\text{array}[\text{string}]\} = \{\text{any}\}
\]
\[
\{\text{array}[\text{int}], \text{array}[\text{bool}], \text{array}[\text{string}], \text{array}[\text{array}[\text{int}]]\} \triangledown_4 \{\text{array}[\text{array}[\text{string}]]\} = \text{array}[\text{any}]
\]

**Figure 3.6** Some examples of using the \(\triangledown_4\) operator to join two type sets

\[
\{\text{bool}\} = \text{depth} = 0
\]
\[
\{\text{array}[\text{int}], \text{bool}\} = \text{depth} = 0
\]
\[
\{\text{array}[\text{int}], \text{array}[\text{string}], \text{array}[\text{bool}]\} = \text{depth} = 1
\]
\[
\{\text{array}[\text{int}], \text{array}[\text{array}[\text{string}]], \text{array}[\text{array}[\text{bool}]]\} = \text{depth} = 1
\]
\[
\{\text{array}[\text{array}[\text{int}]], \text{array}[\text{array}[\text{bool}]], \text{array}[\text{array}[\text{string}]]\} = \text{depth} = 2
\]
\[
\{\text{array}[\text{array}[\text{int}]], \text{array}[\text{array}[\text{bool}]], \text{array}[\text{array}[\text{array}[\text{string}]]]\} = \text{depth} = 2
\]
\[
= \ldots
\]

**Figure 3.7** Some examples to illustrate array depth evaluations

and large type-sets as mentioned in section 3.1.4.

Now, let us see how type unification resolves the non-terminating analysis iteration for the function defined in Fig. 3.3. For this example, consider only the set of types inferred for variable \(\$a\). On each pass through the loop, the transformation operator, \(\Gamma\) would have the effect of returning an \textit{Array} type of the left hand side of Eq. 3.1. The types inferred for \(\$a\) through each iteration is given in Figure 3.8.

The novel thing that happened as a result of the type unification is that beyond \(n = 4^{th}\) iteration, the inferred type for \(a\) does not change. This helps to stop the iterations at that point. Continued iteration would not add any more useful information, nor would it allow the iteration to stabilize. Although we loose precise type information on \(\$a\) as a result of using the widening operator, it doesn't cause us any disadvantage in this case.

To prove the claim that the type unification is sound, let us manually try to infer what the type of \(a\) may be in Fig. 3.3. From simple observation, we can see that it is impossible to precisely determine when the above loop exits. Thus \(a\) could have any array depth. Explicit run-time information is
3.1. ANALYSIS OF METHODOLOGY  

CHAPTER 3. TYPE ANALYSIS FOR PHP

\[
\begin{align*}
type(a)_{n=0} &= \{\text{int}\} \\
type(a)_{n=1} &= \{\text{int}, \text{array[}\text{int}\text{]}\} \\
type(a)_{n=2} &= \{\text{int}, \text{array[}\text{int}\text{]}, \text{array[}\text{array[}\text{int}\text{]}\text{]}\} \\
type(a)_{n=3} &= \{\text{int}, \text{array[}\text{int}\text{]}, \text{array[}\text{array[}\text{array[}\text{int}\text{]}\text{]}\text{]}\} \\
type(a)_{n=4} &= \{\text{array[}\text{any}\text{]}\} = \{\text{any}\} \\
type(a)_{n=5} &= \{\text{array[}\text{any}\text{]}\} = \{\text{any}\}
\end{align*}
\]

**Figure 3.8** Sequence of type sets evaluated for $a$ using the $\nabla_4$ operator

required to precisely determine this. Thus it is indeed a safe approximation to infer that the type of $a$ could be of type any in this case.

It must be noted that the same widening process cannot be applied for types of return values from functions, or for object types. This is because the return type of functions can be dependent on the input arguments. We need to account for the types of the arguments before applying type unification. Likewise, a modified type unification scheme must be used for object types. This is because the object values could easily acquire many different mutable states, owing to the different values that their fields could be set to. The detailed descriptions of how type unifications are applied for objects and function return types are provided in section 3.3.4 and 3.3.7, respectively.

3.1.6 Analysis of constraints

The analysis on this paper is based on a constraint-based approach. Constraints are at the very heart of the analysis and provide a key interface between the target source code (in this case, the target PHP script(s) to analyze) and the actual analysis process. The constraints are generated based on the target script’s CFG. The PHP-Air framework [HK14] provides this capability for us. The CFG, and the generated systems of constraints are all available as Rascal algebraic data types during the analysis. The high-level sequence of operations is given in Fig. 3.9. An example each of the most important analysis components are also given in Appendix-A.

The first step in the type analysis is the generation of the script CFG of the target PHP code. The resulting CFG nodes are consumed to give systems of constraints that are solvable for each node.
Figure 3.9 High level overview of the operations.
3.1. ANALYSIS OF METHODOLOGY

This is represented by $C^*[S_1], ..., C^*[S_N]$ in Fig. 3.9. The CFG pre-processing module uses the CFG and the generated constraints to produce an abstract directed flow-graph which is fed into the type solver. The solver processes this information to give out a typed flow model of the script and a list of warnings. The typed flow model of the script can be viewed using the Graphviz platform [Gra] to see detailed type annotations of each program variables. The warnings provide information on possible errors located by the type analyzer.

The constraints can be understood as a set of behaviors that can be imposed on a PHP statement or expression. They may describe characteristics such as if the unit of code generates a particular kind of value, or data-flow information such as if a unit of code assumes value from a different unit of code. The complete set of constraints is given in Fig. 3.10.

```java
//Defining the algebraic data type: Constraint
public data Constraint=
    yieldType(Lab l1, TypeRoot t)
  |yieldFlow(Lab l1, Lab l2)
  |expectType(Lab l1, TypeRoot t)
  |expectFlow(Lab l1, Lab l2);

//Defining the algebraic data type: TypeRoot
public data TypeRoot =
    typeSet(set[Type] types)]
  |toArray(Lab aLabel)
  |toArraySet(set[Lab] labels)
  |fromArray(Lab l)
  |toArray(Type t)
  |funcCall(str name, list[Lab] params)
  |fromLabels(set[Lab] labels)
  |fromLabel(Lab label)
  |fromVar(Identifier aVar)
  |newObjectInstance(str cName, list[Lab] params)
  |fromObjectProperty(TypeRoot fromExprAtLabel, Identifier fieldVar)
  |fromStaticProperty(str cName, Identifier fieldVar)
  |fromMethodCall(TypeRoot fromExprAtLabel, str fName, list[Lab] params)
  |fromStaticCall(str cName, str fName, list[Lab] params)
  |nullTypeRoot();
```

Figure 3.10 Constraints used in this analysis
From Fig. 3.10 we can see that there are four kinds of constraints in our system. Each type of constraint is defined as a unique constructor function for the algebraic type $\text{Constraint}$. Each constraint is also related to a label: $l1$, and a second parameter that could either be the algebraic type: $\text{TypeRoot}$, or another label: $l2$. Together, these constraints describe the rule by which to compute type values for any expression in the program. A brief description of each of the constraint types is provided below.

- The constraint $\text{yieldType}$ describes that the expression with the label: $l1$ generates a particular type of value in the program. If the type of the value is directly observable, such as is the case with literals, then $t$ is instantiated with the constructor function: $\text{typeSet}$, and the parameter $\text{types}$ initialized to the known types. Very often though, that is not the case. Expressions may be formed from sub-expressions, and them in turn from other sub-expressions. The constraints generation task involves analyzing these dependencies and forming the appropriate tree of constraints to represent this dependency. In such cases, for a constraint of type $\text{yieldType}$, the parameter $t$ is instantiated to one of the constructor functions that best describes the expression at label $l1$.

- The constraint of type $\text{expectType}$ describes the kind of types to expect from the expression at label $l1$ in the program. These constraints play an important role in the analysis. They form the basis for checking if the computed types for expressions are compatible in the context that they appear in. If this compatibility check fails, the analysis produces a warning that can alert the developer to this potentially erroneous site. Unlike constraints of type $\text{yieldType}$, the second parameter for these constraints, i.e. $t$, can only assume the $\text{typeSet}$ constructor function. This is because the expected types are assumed to be known beforehand. They are not meant to be computed.

- The constraints of types $\text{expectFlow}$ and $\text{yieldFlow}$ both specify the flow of information from the expression at $l2$ to $l1$. In the case of the former, it specifies that each expected type for the expression at label $l2$ is also expected for the expression at label $l1$. In the case of the latter, the constraint specifies that the types for $l2$ must be computed as the set of types for $l1$.

In the rest of the thesis, we use the following set of notations to describe the constraints. If an expression at $l$ yields a set of types, we write $\{\tau_1, \tau_2, \ldots, \tau_n\} \subseteq \text{type}(l)$. Most of the time, an expression yields a single type. We express this notationally as $l := \tau$. Likewise, we use the notation $\{\tau_1, \tau_2, \ldots, \tau_n\} \subseteq \text{type}(l)$ to denote the set of expected types for an expression at $l$. The notation
3.2 Constraints Generation

3.2.1 Literals

A literal generates a single constraint of the type \( l := \tau \), where \( \tau \) is any concrete type from Fig. 3.2. All the possible constraints generated by various literals in PHP are given in Fig. 3.11. These constraints were taken directly from [Cam09], and is adequate for capturing the generation of types for literals in PHP.

3.2.2 Expressions and operators

An expression may generate a number of different constraints. They are enumerated in Fig. 3.12 and Fig. 3.13. Expressions that involve an operator generate additional constraints of the type \( l := \tau \), since symbolic operators expect a certain type of value for its operands in PHP. We can do this only because it is not possible to overload symbolic operators such as +, -, or [ ] in PHP. Thus, it is fairly
3.2. CONSTRAINTS GENERATION

accurate to say that the type of value expected is respectively: numeric, string, and an array in the given case of operators.

Most of the constraints in Fig. 3.12 and Fig. 3.13 were taken from [Cam09]. The constraints that were added are fromLabel, fromLabels, and fromVar. These constraints are critical pieces that tie in the generation of types to certain expressions of interests within a PHP statement. They are used for expressions whose concrete types cannot be evaluated at compile-time, such as with variables and function call expressions. When these constraints are encountered, the type analyzer defers type inference until the actual solution process when all runtime information is available. This makes it possible to generate much more accurate types for the expressions of interest.

3.2.3 Statements

Statements are composed of expressions, and do not generate or hold values by themselves. However, statements may provide the context in which an expression is used. This contextual information could be exploited to provide expected type information of certain expressions. We list them in Fig. 3.14.

We would like to point out the following:

- `toArray(l)` returns the type: Array, of the set of types obtained from label l. For e.g. if `type(l)` is `{Int(),String()}`, then `toArray(l)` yields the set of types: `{Array(Int()),Array(String())}`. Likewise, `toArray(any)` returns the type `Array(any)` for the expression that yields this constraint.

- `toArraySet(l_1,…,l_n)` returns the Array type of each the of type obtained for labels: $l_1$ through
3.2. CONSTRAINTS GENERATION

CHAPTER 3. TYPE ANALYSIS FOR PHP

\[
C^T\left[(e^1_l)^l\right] = \{ l_1 := fromLabel(l_1) \}
\]

\[
C^T\left[(e^1_l, \ldots, e^n_l)^{l_{n+1}}\right] = \{ l_{n+1} := fromLabels(l_1, \ldots, l_n) \}
\]

\[
C^T\left[\{ e_l^i \}^l\right] = \{ l_1 \equiv toArray[\text{any}], l \equiv fromArray(l_1), \{ \text{int, string} \} \subseteq \text{type}(l_2) \}
\]

\[
C^T\left[\{ e^i_l = e^j_l \}^l\right] = \{ l_1 \leftarrow l_2, l_\leftarrow l_2 \}
\]

\[
C^T\left[\{ \text{var}^i_l \}^l\right] = \{ l_1 := fromVar(var) \}
\]

\[
C^T\left[\{ \text{operator}^i_l \}^l\right] = \{ l_1 \equiv \text{int, } l := \text{int}, C^T\left[\{ \text{operator}^i_l \}^l\right] = \{ l_1 \equiv \text{int, } l := \text{int} \}
\]

\[
C^T\left[\{ \text{operator}^i_l \}^l\right] = \{ l_1 \equiv \text{bool, } l := \text{bool} \}
\]

\[
C^T\left[\{ \text{operator}^i_l \}^l\right] = \{ l_1 \equiv \text{num, } l := \text{num} \}
\]

\[
C^T\left[\{ \tau \}^l\right] = \{ l := \tau \} \text{ where } \tau \in \text{base type}
\]

\[
C^T\left[\{ \text{count}^i_l \}^l\right] = \{ l := \text{int, } l_1 \equiv toArray[\text{any}] \}
\]

\[
C^T\left[\{ \text{array}^i_l \}^l\right] = \{ toArray[\tau] : \tau \in \text{type}(l_1) \mid i \in \{1 \ldots n\} \subseteq \text{type}(l) \}
\]

**Figure 3.12** Constraints generated by basic expressions in PHP

\[
C^T\left[\{ e^i_l \oplus e^j_l \}^l\right] = \{ l_1 \equiv \text{num, } l_2 \equiv \text{num, } l := \text{num} \} \text{ where } \oplus \in \{ +, -, \ast, /, \% \}
\]

\[
C^T\left[\{ e^i_l \ast e^j_l \}^l\right] = \{ l_1 \equiv \text{string, } l_2 \equiv \text{string, } l := \text{string} \}
\]

\[
C^T\left[\{ e^i_l \ast e^j_l \}^l\right] = \{ l_1 \leftarrow l_2, l := \text{bool} \}
\]

\[
C^T\left[\{ e^i_l \ast e^j_l \}^l\right] = \{ l_1 \leftarrow l_2, l := \text{bool} \}
\]

\[
C^T\left[\{ e^i_l \oplus e^j_l \}^l\right] = \{ l_1 \equiv \text{num, } l_2 \equiv \text{num, } l := \text{bool} \} \text{ where } \oplus \in \{ <, >, \leq, \geq \}
\]

\[
C^T\left[\{ e^i_l \oplus e^j_l \}^l\right] = \{ l_1 \equiv \text{bool, } l_2 \equiv \text{bool, } l := \text{bool} \} \text{ where } \oplus \in \{ \&\& , || \}
\]

\[
C^T\left[\{ e^i_l \ast e^j_l : e^j_l \}^l\right] = \{ l_1 \equiv \text{bool, } l_2 \subseteq \text{type}(l), l_3 \subseteq \text{type}(l) \}
\]

**Figure 3.13** Constraints generated by basic operators in PHP
3.2. CONSTRAINTS GENERATION

\[ C^l \left[ \text{echo } e_1^{l_1}, \ldots, e_n^{l_n} \right] \times \{ l_i \equiv \text{string} \mid i \in \{1, \ldots, n\} \} \]

\[ C^l \left[ \text{fromArray}(l) \right] = \{ l \leftarrow \text{toArray}(l), l_2 \subseteq \{ \text{int, string} \} \} \]

\[ C^l \left[ \text{toArray}(l) \right] = \{ l_1 \leftarrow \text{fromArray}(l_2), l_2 \equiv \text{toArray}(\text{any}), l_3 \subseteq \{ \text{int, string} \} \} \]

\[ C^l \left[ \text{toArray}(l) \right] = \{ l_1 \leftarrow l_3, l_2 \subseteq \{ \text{int, string} \}, l_4 \subseteq \{ \text{int, string} \} \} \]

\[ C^l \left[ \text{toArray}(l) \right] = \{ l_0 \leftarrow l, l := \text{toArraySet([l_1, \ldots, l_n])} \} \]

\[ C^l \left[ \text{toArray}(l) \right] = \{ l_0 \leftarrow l, \{ \text{type}(l_2), \text{type}(l_3) \} \subseteq \text{type}(l), l_1 \equiv \text{bool} \} \]

**Figure 3.14** Constraints generated by basic statements in PHP

\[ l_n \text{. For example, for the php expression, } \$a = \text{array}(1, \"a\", \text{true}) \text{, the resulting type assigned to } \$a \text{ would be: } \{ \text{Array(Int())}, \text{Array(String())}, \text{Array(Bool())} \}. \]

- *fromArray(l)* acts in the opposite direction of *toArray*. It returns the base type of the array in expression at label *l*. If the expression in label *l* doesn't represent an array type, then a warning is produced, and a *Void* type is assigned to that expression.

Fig. 3.15 provide additional constraints generated by simple self-assignment statements in PHP.

\[ C^l \left[ \text{echo } e_1^{l_1}, e_2^{l_2} \right] = \{ l_1 \leftarrow \text{num}, l_2 \equiv \text{num} \text{ where } \leftarrow \text{\{+ =, -, * =, / =, % =\}} \}

\[ C^l \left[ \text{echo } e_1^{l_1}, e_2^{l_2} \right] = \{ l_1 \leftarrow \text{string}, l_2 \equiv \text{string} \text{ where } \leftarrow \text{\{. =, =\}} \}

\[ C^l \left[ \text{echo } e_1^{l_1}, e_2^{l_2} \right] = \{ l_1 \leftarrow \text{bool}, l_2 \equiv \text{bool} \text{ where } \leftarrow \text{\{& =, | =\}} \}

**Figure 3.15** Constraints generated by self-assignment statements in PHP

Control structures allow for alternate execution paths along the code. We need to analyze each of these branches and merge type information available from each execution trace. In the meantime, the control statement itself generates few additional constraints. We list them in Figures 3.16, and
The constraints in Fig. 3.14 - Fig. 3.15 were also all taken from [Cam09]. We found these constraints to be adequate for generating types for the involved cases, and so we do not have new constraints to add.

\[
\begin{align*}
\mathcal{C}_l\left[\left(\text{if } e_1^{l_1} \text{ } S_1^{l_2} \right)\right] & = \{l_1 \equiv \text{bool}\} \\
\mathcal{C}_l\left[\left(\text{if } e_1^{l_1} \text{ } S_1^{l_2} \text{ else } S_2^{l_3} \right)\right] & = \{l_1 \equiv \text{bool}\} \\
\mathcal{C}_l\left[\left(\text{switch } e_1^{l_1} \text{ case } v_1^{l_1}, \ldots, v_n^{l_n} \right)\right] & = \{l_\epsilon \leftarrow l_1 \mid \epsilon \in \{1, \ldots, n\}\}
\end{align*}
\]

Figure 3.16 Constraints generated by control statements in PHP

\[
\begin{align*}
\mathcal{C}_l\left[\left(\text{while } e_1^{l_1} \text{ } S_1^{l_2} \right)\right] & = \{l_1 \equiv \text{bool}\} \\
\mathcal{C}_l\left[\left(\text{do } S_1^{l_2} \text{ while } e_1^{l_1} \right)\right] & = \{l_2 \equiv \text{bool}\} \\
\mathcal{C}_l\left[\left(\text{for } dcl^{l_d} ; e_1^{l_1}, \ldots, e_n^{l_n} ; incs^{l_i} \text{ } S_1^{l_1} \right)\right] & = \{l_k \equiv \text{bool} \mid k \in \{1, \ldots, n\}\}
\end{align*}
\]

\[
\begin{align*}
\mathcal{C}_l\left[\left(\text{foreach } e_1^{l_1} \text{ as } e_2^{l_2} \text{ } S \right)\right] & = \{l_1 \equiv \text{toArray}[\text{any}], \ l_2 \leftarrow \text{fromArray}(l_1)\} \\
\mathcal{C}_l\left[\left(\text{foreach } e_1^{l_1} \text{ as } e_2^{l_2} \Rightarrow e_3^{l_3} \text{ } S \right)\right] & = \{l_1 \equiv \text{toArray}[\text{any}], \ l_3 \leftarrow \text{fromArray}(l_1), \ l_2 \subseteq \{\text{int, string}\}\}
\end{align*}
\]

Figure 3.17 Constraints generated by looping structures in PHP
3.3 Solving the constraints

Once the constraints are formed, we are ready to proceed with the actual analysis. The result of this analysis is the derivation of a set of types for each program point $P$, in the given PHP script. The algorithm follows the program flow as specified by the CFG generated for the script. This program flow is specified at the statement level. Within each statement and where applicable, the types for expressions and sub-expressions are evaluated recursively. This means that the algorithm will drill down through expressions until a concrete set of types can be inferred. A concrete set of types is yielded by expressions that either directly hold literals, or reference program entities such as variables and functions that generate a value. The details of this process is enumerated in the rest of this section.

3.3.1 Definitions

A $TypeEnvironment$ is used as an alias to describe a map of $<variable, TypeRoot>$ data pairs. It is essentially defined in the analysis tool as the follows (actual Rascal code):

\[
\text{public alias TypeEnvironment} = \text{map} [\text{Identifier}, \text{TypeRoot}];
\]

A unique such map is instantiated for each statement in the script. The $TypeEnvironment$ data structure holds information on the types of all the variables in the script when the program execution has reached that statement. Three instances of the data type $TypeEnvironment$ are defined for each program statement (node). They are as follows:

- The $entry$ variable stores the information of the type-roots of all the [PHP] application variables as execution prepares to enter a particular program statement. It capture the state of the program right before a particular statement is executed.

- The $expect$ variable stores information on the expected type-roots of all the application variables as a result of processing the content of the current program statement. This data plays an integral role in our analysis and helps us capture any unintended or incompatible operations.
3.3. SOLVING THE CONSTRAINTS

The exit variable stores the type-roots of all the application variables that resulted after the current statement was executed. A statement usually has the effect of assigning a value to one particular variable in the program. Since this tool is relatively new, we support single assignment expressions only at the moment. If this assignment fails for any reason such as assigning from a non-defined function, uninitialized variable, or by using the subscription ([ ]) operator on a non-array value, then the variable is assigned a type-root value: 
\[\text{typeSet} (\text{Void}())\]. For all other situations, the variables assume a type-root constructor of type: 
\[\text{typeSet} (\{\tau_1, \ldots, \tau_n\})\], where \(\{\tau_1, \ldots, \tau_n\}\) is the set of types inferred after solving the constraints for this statement.

We now proceed to define a new operator that is used to combine two type environments. We call that to be the unification operator and denote it using \(\sqcup\). This operator takes in two operands each of type \(\text{TypeEnvironment}\) and returns a value of the same type. It uses the delta operator \(\nabla\) defined previously (Fig. 3.5) to combine the type-sets of each program in the given type environments.

\[\sqcup : (TEnum)_A \times (TEnum)_B \rightarrow (TEnum)_C\]

Another important definition to be familiar with is the state of a node. This is defined as:

\[\text{state}(\text{node}) : P_{\text{node}} \rightarrow (TEnum)_{\text{entry}} \times (TEnum)_{\text{exit}} \times (TEnum)_{\text{expect}}\]

and is the collection of the entry, exit, and expect type environments for the given node. The result of this analysis, then, is a collection of state instances for the given PHP script, where each state gives the state of the application at that particular statement. Collectively, we represent this as \(\text{state}^*\).

And lastly, we introduce the concept of an analysis environment. An analysis environment can be considered as a collection of resources that enable type analysis through a certain piece of PHP code. This collection contains, among others, data structures such as systems of constraints, an abstract directed flow-graph, and dictionaries of objects, function templates, and class types defined in the PHP application. In our work, separate analysis environments are created for the main script, each function definition, and any script included during the analysis.
3.3. SOLVING THE CONSTRAINTS

CHAPTER 3. TYPE ANALYSIS FOR PHP

```
1: procedure PROPAGATE(P*, ∇, F₀)
2: 3:    // Define a worklist stack
4:    Stack<Edge> W=new Stack<Edge> ;
5: 6:    // Step 1: Initialization
7:    for all p ∈ P* do
8:        state(p)={φ, φ, φ}
9:    end for
10:    for all Edge e ∈ F₀.edges do
11:        W.push(e)
12:    end for
13: 14:    // Step 2: Iterate
15:    while W ≠ nil do
16:        edge := W.pop(); node := edge.to;
17:        (entry,exit,_) := state(node);
18:        constraints := (node)constraints
19:        <new_set,warnings> := transfer_types(entry,constraints,∇);
20:        exit' := exit ∪ new_set;
21:        if exit' ≠ exit || notVisited(node) then
22:            exit := exit' ;
23:            for all node ∈ P* : (node, node') ∈ F do
24:                edge' = connect(node, node');
25:                W.push(edge');
26:                (entry', _, _) := state(node');
27:                entry' := entry' ∪ exit' ;
28:            end for
29:        end if
30:    end while
31:    return <state*,warnings> ;
32: end procedure
```

Figure 3.18 Propagate type information through a php script
3.3. SOLVING THE CONSTRAINTS

3.3.2 The basic algorithm

Fig. 3.18 outlines the basic steps taken to infer types for a script without any function calls and include statements. The methodology is based on the modified version of the worklist algorithm in [FNH05]. The method propagate is at the heart of the analysis and returns the entry, exit, and expected types for all the variables appearing locally in the script. We briefly outline the steps involved.

The method in Fig. 3.18 takes a list of three parameters. The first parameter $P_\ast$ stands for the set of analysis nodes. Each node corresponds to a statement in the PHP script. The last two parameters: $\triangledown$ and $F_0$ stand for the unification (widening) operator and the program entry point respectively.

The 1$\text{st}$ step involves an initialization process that initializes the state for each program node to empty sets. During this state, no type information is known or assumed about any of the program points. We also take the edges $(e)$ originating from $F_0$ and push them to the worklist stack $W$.

Following this, we enter the 2$\text{nd}$ and most important phase of the analysis. An edge is popped from the worklist stack $W$, and it is used to get the node that this edge is connected to (line: 16). Following that, the set of constraints for the current node is retrieved. The function transfer_types is consequently called passing the constraints as one of the arguments (line 19). This method returns a tuple of values, i.e. the state of the node, and the list of any warnings that may have resulted from analyzing the content of this statement.

The value new_set (line: 19) denotes the new type information available after solving the constraints for this node. This new set of type information is combined with the type information that we already had (exit) from the previous iteration. This process of combining type information from different iterations is performed using the unification operator ($\sqcap$) and we denote the resulting type environment as exit$'$. After exit$'$ has been calculated, we compare to see if it is equal to exit, which is the state of the node from the previous iteration. If they are equal, we do not push type-analysis beyond this node. Instead, we proceed to analyze the next edge (if any) on the worklist $W$ (line: 16). This is repeated until there are no more nodes left to be analyzed for the analysis.

A more interesting case happens when exit$'$ is different from exit (lines: 21-29). In this case, new type information was available from the currently analyzed node. This information must be propagated to all the nodes that are connected to the current node in the program flow. To achieve this, all the
edges emanating from the current node is pushed to the worklist \( W \). In addition to that, \( \text{exit} \) is also combined with the entry environment of each of the destination nodes using the \( \sqcup \) operator.

### 3.3.3 The transfer_type method

Fig. 3.19 shows the algorithm for the transfer_type function. In the 1\(^{st}\) step, the resolve_dependencies method is called that essentially solves the system of constraints to infer a set of types for each identifier in the statement. This method call also returns a set of warnings that may have resulted from the call.

```plaintext
1: function TRANSFER_TYPE(entry, constraints, \( \nabla \) )
2:
3:   types := entry ;
4:
5:   //Step 1: resolve type-flow dependencies and solve system
6:   <ccs, warnings> = resolve_dependencies(constraints, entry);
7:
8:   //Step 2: Update type information on identifiers
9:   for all cc \( \in \) ccs do
10:      type_set := cc.types ;
11:      if cc.\( l \) \( \in \) Id then
12:         types(l) := type_set ;
13:      end if
14:   end for
15:
16:   //Step 3: Generate warnings using expected types
17:   warnings += generateWarnings(ccs);
18:
19:   return <types, warnings> ;
20: end function
```

**Figure 3.19 Outline of the transfer_type function**

In the 2\(^{nd}\) step (lines: 9-14), the solved constraints (ccs) are used to assign new set of types to any identifiers that were affected. And finally, the 3\(^{rd}\) step involves generating additional warnings by comparing the new type information with the expected set of types. The details of this process is
3.3. SOLVING THE CONSTRAINTS

illustrated in Fig. 3.23. When this is done, the method returns.

For example, consider the first statement in the example script (Fig. A.1). This statement, labelled $l_3$ is as follows:

\[
(a^l = 7^h)^h;
\]

and, generates the constraints given by $C^h$:

\[
\begin{align*}
& l_1 \Leftarrow l_2, \quad (3.2) \\
& l_3 \Leftarrow l_2, \quad (3.3) \\
& l_2 := \text{int.} \quad (3.4)
\end{align*}
\]

Constraint 3.2 is generated because the equality sign suggests the flow of types from label $l_2$ to $l_1$. Constraint 3.3 suggests the equivalent idea, i.e. the expression labelled by $l_3$, which is the assignment statement, generates a value of type as specified by $l_2$. The rules for the generation of both these constraints are defined in Fig. 3.12. Finally, constraint 3.4 indicates that a value of type \text{int} is generated from this expression. This is a straightforward inference since the value at $l_2$ is an integer literal. The rule for this is defined in Fig. 3.11.

The processed constraints $ccs$ from the \textit{resolve_dependencies} function looks as follows:

\[
ccs: \quad l_1 := \text{int}, l_3 := \text{int}, l_2 := \text{int.} \quad (3.5)
\]

In the 2\textsuperscript{nd} step of this method, it detects that $l_1$ directly references the program variable $a$ in this statement. The information from 3.5 is then used to assign the type(s) for $a$ to the \text{int} type.

3.3.4 The \textit{resolve_dependencies} method

Fig. 3.20 shows the basic steps involved in the \textit{resolve_dependencies} method. This method is essentially the core solver that returns a set of constraints that has no data flow dependencies and all the constraints reference a concrete type root, i.e. a TypeRoot data type of form: $\text{typeSet}([\tau_1, \ldots, \tau_n])$. It takes in the entry type-environment and a set of constraints as the input argument and makes a series of deductions using some helper methods. The 1\textsuperscript{st} step involves some initializations. The other two parts are a bit involved will be briefly outlined below.
1: function RESOLVE_DEPENDENCIES(entry,constraints )
2: 3: //Step 1: Some initializations
4: 5: to_return = 0;
6: new_constraints = 0;
7: warnings = 0;
8: 9: //Step 2: Resolve any type-flow constraints
10: for all cc ∈ constraints do
11: 12: if cc is $l_1 \leftarrow l_2$ then
13: 14: new_constraints += \{ $l_1 := resolve\_flow(l_2, constraints)$ \} ;
15: 16: else
17: 18: new_constraints += cc;
19: end if
20: end for
21: 22: //Step 3: Get the concrete types for each label
23: for all cc ∈ new_constraints do
24: 25: if cc is $l_i := \tau$ then
26: 27: <concrete_type,some_warnings>=get_concrete_type_from(\tau);
28: 29: to_return += \{ $l_i := concrete\_type$ \};
30: 31: warnings += some_warnings;
32: 33: else
34: 35: to_return += cc;
36: end if
37: end for
38: 39: return <to_return,warnings> ;
40: end function

Figure 3.20 Outline of the resolve_dependencies function
In the 2\textsuperscript{nd} step (lines: 9-15), the method tries to resolve any constraints that represent a type-flow. This means that any constraints of the type: \( l_i \Leftarrow l_{i+1} \) will be flattened out so that each constraint of this type will be changed to the form: \( l_i := \tau \), where \( \tau \) is any of the possible constructor values of the ADT \textit{TypeRoot}. The method: resolve\_flow is primarily responsible for this. Note that all other constraints, including the expected type-flow constraints, i.e. \( l_i \Leftarrow l_{i+1} \) is kept unchanged at this step.

In the 3\textsuperscript{rd} step (lines: 18-26), each of the constraints from the 2\textsuperscript{nd} step are analyzed to infer a concrete set of types. The steps to this are outlined in Fig. 3.22. The key player in this part of the analysis is the function: \textit{get\_concrete\_type\_of} that takes a label, a set of constraints, and the entry type environment as arguments and returns a concrete type root for that label.

This function (Fig. 3.22) may undergo recursion to resolve what the actual set of types of an expression may be. This is usually required when generating types for a non-literal expression, or any expression that generates a value whose type is not known at compile-time. This was briefly discussed in section 3.2.2.

As a running example to illustrate some of the algorithmic steps for this method, we consider the statement marked with label \( l_{32} \) from the example script. The statement is as follows:

\[
(sz_{l_{28}} = \text{array}(x_{l_{29}}, y_{l_{30}}))_{l_{31}}^{l_{32}}
\]
3.3. SOLVING THE CONSTRAINTS

Chapter 3. Type Analysis for PHP

function GET_CONCRETE_TYPE_OF(l_k, constraints, entry) {
    ab := cc \tau where cc \in constraints, and cc.l = l_k;
    switch ab do
    case fromFunc(name_f, [l_1, ..., l_n]):
        return type_set\{analysis_f([[t_i|t_i \leftarrow get_concrete_type_of(l_i, constraints, entry)]])\};
    case toArray(l_i):
        return type_set\{[Array(t_i)|t_i \leftarrow get_concrete_type_of(l_i, constraints, entry)]\};
    case fromArray(l_i):
        return type_set\{[t_i|Array(t_i) \leftarrow get_concrete_type_of(l_i, constraints, entry)]\};
    case fromLabels\{\{l_1, ..., l_n\}\}:
        return \{\sum^{N}_{j=1} (\nabla \alpha)_i|t_i \leftarrow get_concrete_type_of(l_i, constraints, entry)\};
    case fromLabel(l_i):
        return \{[get_concrete_type_of(l_i, constraints, entry)]\};
    case fromVar(var(name)):
        return entry[var(name)];
    case default:
        return ab;
end function

Figure 3.22 Outline of the get concrete type root function
The raw set of constraints generated by this statement is,

\[
\begin{align*}
  l_{32} & \leftarrow l_{31}, \quad (3.6) \\
  l_{31} & \leftarrow toArraySet(l_{29}, l_{30}), \quad (3.7) \\
  l_{29} & \leftarrow fromVar("x"), \quad (3.8) \\
  l_{30} & \leftarrow fromVar("y"), \quad (3.9) \\
  l_{28} & \leftarrow l_{31}. \quad (3.10)
\end{align*}
\]

Constraints 3.6 and 3.10 represent that the expressions at \( l_{28} \) and \( l_{32} \) both acquire the types generated by the right-hand-side of the assignment expression, i.e. \( l_{31} \) (also see 3.2 and 3.3). Constraint 3.7 informs that the type resulting from expression at \( l_{31} \) is actually an array, and the rule for this is specified in Fig. 3.14. Finally, constraints 3.8 and 3.9 both represent that the information of types for expressions at \( l_{29} \) and \( l_{30} \) are to be taken from program variables \($x$ and \$y$, respectively. Together, these constraints form the complete set of constraints generated for this statement.

In the 2\textsuperscript{nd} step of this method (i.e. \texttt{resolve_dependencies}), the call to the \texttt{resolve_flow} method removes all constraints of the type \(. This is an important step, since it directly associates the involved expressions (in this case, expressions at \( l_{28} \) and \( l_{32} \)) to the source of the types (i.e. \( l_{31} \)). The resulting set of constraints as a result of this flattening process is:

\[
C^{l_{32}} : \quad l_{32} \leftarrow toArraySet(l_{29}, l_{30}), \\
l_{31} \leftarrow toArraySet(l_{29}, l_{30}), \\
l_{28} \leftarrow toArraySet(l_{29}, l_{30}), \\
l_{29} \leftarrow fromVar("x"), \& \\
l_{30} \leftarrow fromVar("y")
\]

The 3\textsuperscript{rd} and final step of this method involves deducing the concrete types of each constraint from above. Using the \texttt{get_concrete_type_of} method (Fig. 3.22), it is possible to deduce the final set of types for each constraint. We further ensure that this effort is not duplicated, i.e., once the concrete types for the constraint: \( \texttt{toArraySet}(l_{29}, l_{30}) \) is evaluated, this result is temporarily stored in an internal map. The final set of solved constraint for this system may look as below:
of course, assuming that the types of $x$ and $y$ supplied in the function call were of the `int` and `str` types respectively.

This function may also return a list of warnings. For clarity of the steps, we do not explicitly show this in Fig. 3.22. Any time that the function fails to find a concrete set of types for a label, an appropriate warning is generated. It may fail for a number of reasons, such as trying to derive values from a variable that has not been initialized yet, a call to a function whose definition is unavailable, or assigning types from a function that does not return any value. The list of generated warnings generated is aggregated and allowed to bubble back to the `transfer_type` function. This is ultimately made available to the users for recommended reviews.

### 3.3.5 Generating warnings

The process of generating warnings is an important application of this tool. This, along-with a typed flow-model of the script is the net output of the type analyzer developed in this work. The latter is largely an internal, non-user facing component. The typed flow-model of the script contains the set of types inferred for each variable, at each program statement in the program, potentially with verbose information and suited ideally only for debugging.

Warnings are generated in virtually all the different methods participating in the type analysis. In Fig. 3.23, we present a class of warnings generated by using the second class of constraints we've ignored so far. These are warnings of the `expected` types, and serve to ensure that types generated conform to the types expected for expressions whenever such expectations can be placed.

In the $1^{st}$ step (lines: 6-15), for each constraint of the `expected` type, it is ensured that the derived types are compatible. If this check fails, a warning of type `IllegalTypeCoercions` is issued with the details of the error. In the $2^{nd}$ step of this function (lines: 19-26), all constraints of type: $l_i \leftarrow l_j$ is checked to ensure that each derived types for $l_i$ is compatible to the derived type for expression at $l_j$. Failure to meet this check results in a type coercion warning as in the $1^{st}$ step.
3.3. SOLVING THE CONSTRAINTS

CHAPTER 3. TYPE ANALYSIS FOR PHP

function GENERATE_WARNINGS(constraints, entry)
    warnings = \emptyset;
    //Step 1: Iterate through all expectType constraints
    for cc ∈ constraints, cc is l ⊆ τ do
        //get the type(s) evaluated for given label
        cc' = cc' ∈ constraints, cc' is l := τ & cc.l = cc'.l;
        //assert that expected and got types are compatible
        if !compatible(cc.τ, cc'.τ) then
            warnings += illegalTypeCoercion(msg);
            end if
    end for
    // Step 2: Iterate through constraints that specify
    // both labels have to have same types
    for cc ∈ constraints, cc is l₁ ↔ l₂ do
        cc'₁ = cc'₁ ∈ constraints, cc'₁ is l := τ & cc.₁.l = l₁;
        cc'₂ = cc'₂ ∈ constraints, cc'₂ is l := τ & cc.₂.l = l₂;
        if !compatible(cc'₁.τ, cc'₂.τ) then
            warnings += illegalTypeCoercion(msg);
            end if
    end for
    return warnings;
end function

Figure 3.23 Outline of the function that generates warnings based on incompatible type coercions
3.3.6 Functions and function calls

A function definition is treated as a top-level statement in our analysis, and a separate *analysis environment* is spawned for each function definition available in the current PHP application. A function definition itself does not yield any constraints. It is only when a function call site is encountered, that a unique set of constraints are generated for the call expression. A function call expression is treated as any other expression that may generate some value. The set of constraints generated in a function call expression is listed in Fig. 3.24.

\[
C[l_0 \leftarrow \text{fromFunc}\{\text{name}, \{l_1, \ldots, l_n\}\}] = \{l_{n+1} := \text{fromFunc}\{\text{name}, \{l_1, \ldots, l_n\}\}\}
\]

*Figure 3.24* Constraints generated by a function call statement in PHP

When the constraints are being solved, the constraints of type *fromFunc* cause the analysis to propagate through the target function’s body. The type of values generated by the function call is calculated as the aggregation of the types of values appearing in all the return statements in the function body. We use the widening operator \((\nabla_q)\) to calculate the final set of types returned. This is expressed notationally in Fig. 3.25.

\[
\text{(result)}_f = \sum_{1}^{N} (\nabla_q)\{\text{fromLabel}(l_i) \mid \text{return}_i(e^i) \leftarrow f_{\text{block}}\}
\]

*Figure 3.25* Determining type-set of values from function

We do not always propagate analysis through the functions. We use a form of function templates structure to map the return type of each function call expression to a function call signature. This lets us reuse the type definition for function results from previous computations.
3.3.7 Function templates

The approach of using function templates is fundamentally based on the Cartesian Product Algorithm (CPA) [references*]. In this algorithm, a series of function templates are created based on the types of the arguments passed during a function call. These templates are maintained throughout the analysis and serve as re-usable analysis components. Each template is associated with a single kind of returned value from the function. Each function call may involve the evaluation of several templates. In our work, when two or more templates are involved, the final resulting type is combined using the widening (\(\nabla\)) operator.

Consider the labeled function call expression

\[
\left( \text{func\_call}(arg^1_1, arg^2_2, ..., arg^K_K) \right)^{k+1}
\]

where the types of the arguments are respectively,

\[
type(arg_1) = \Gamma_1 = \left\{ \tau^1_1, ..., \tau^1_{N_1} \right\},
\]
\[
type(arg_2) = \Gamma_2 = \left\{ \tau^2_1, ..., \tau^2_{N_2} \right\},
\]

....

\[
type(arg_K) = \Gamma_K = \left\{ \tau^K_1, ..., \tau^K_{N_K} \right\},
\]

where \(\tau\) is a concrete type, i.e. any one in the set: \{int, float, ..., any\}. The creation of function templates involve taking the cartesian product of the set of types of each argument. The result of this is a relation, where each element is an ordered set corresponding to a unique combination of the supplied types. We refer to an ordered set as a tuple interchangeably in this text since they both mean the same mathematical object.

The cartesian product: \(\Gamma_1 \times \Gamma_2 \times ... \Gamma_K\) gives the relation: \(\Pi^K_1\)

\[
\Pi^K_1 = \left\{ \langle \tau^1_i_1, \tau^2_i_2, ..., \tau^K_i_K \rangle | i_1 \in \left\{ \tau^1_1, ..., \tau^1_{N_1} \right\}, i_2 \in \left\{ \tau^2_1, ..., \tau^2_{N_2} \right\}, ..., i_K \in \left\{ \tau^K_1, ..., \tau^K_{N_K} \right\} \right\}
\]

where each \(\langle \tau^1_i_1, \tau^2_i_2, ..., \tau^K_i_K \rangle\) represents one unique combination of the concrete types from the passed arguments: \(arg_1, arg_2, ..., arg_K\). After this set of tuples is evaluated, we proceed to analyze...
the return types from the function taking each of this tuple as the input argument for the type analysis. The final type result for this function call is then evaluated by widening the type result obtained for each ordered type set, i.e.

$$\text{type}(l_{K+1}) = \sum_{m \in \Pi_i^K} \left( \text{type}(\tau_{i_1}^1, \tau_{i_2}^2, \ldots, \tau_{i_K}^k) \right)_m$$

When the same function is called for a different set of arguments later in the script, analysis proceeds similarly by calculating the cartesian product of the arguments’ type sets. This time however, only those tuples whose return types are not available will be calculated in the type analysis for the function. This lets us avoid redundant analysis in situations that do not really produce any new type information.

One of the primary advantages of using this approach for typing functions is that no approximations need to be made for the types of the arguments being passed. Since the type information for any combination of the input types is stored during the analysis, redundant work is avoided by doing the same for later function calls involving the same input types. Finally, it is quite simple to both implement and understand this approach.

An example of the constraints arising from a function call is demonstrated in the example script. The statement:

$$\text{cl22} = \text{some_func}(\text{al23}, \text{bl24})$$

gives rise to the following set of constraints,

$$l_{25} := \text{funcCall}(\text{"some_func"}, [l_{23}, l_{24}]), l_{23} := \text{fromVar("a")}, l_{24} := \text{fromVar("b")}, l_{22} \leftarrow l_{25}, l_{26} \leftarrow l_{25}$$

Now let us suppose that the deduced types of $a$ and $b$ are \{int\} and \{int, str\} respectively. Then, taking the cross product of these sets gives us the following relation,

$$\Pi = \{(int, int), (int, str)\}$$

Type analysis through the function proceeds in two steps. In the first, both the formal parameters
acquire the types \textit{int}. The return type of the function evaluates to \textit{array(int)} in this case. In the second step, the formal parameters acquire the types \textit{int} and \textit{str} respectively. The inferred type for the function using these input argument types is \textit{array(int,str)}. The final inferred type is taken by combining these types using the widening operator. This results in

\[
\text{typeOf}(\$c) = \text{array\{int\}} \sqcup \text{\{array(int, array(str))\} = \text{\{array(int), array(str)\}}
\]

### 3.3.8 Handling recursive function calls

Type analysis involving recursion required some special considerations. The first step in tackling recursion is to build some form of a function call sequence graph. The idea is that such a graph can help us detect recursion because any recursive call sequences would lead to a cyclic graph structure.

We do not exactly use a call sequence graph in our analyzer. Instead, what we build is a function call context stack. This stack structure stores the called functions' signatures (name and arguments type) in the order in which the call expressions were encountered in the application. Before type analysis begins in any function body, the type checker queries the stack to see if the function name and associated type signature is already present in this stack. If the query returns false, then analysis propagates normally in the function. If the query returns true, then the propagation mode is marked as \textit{recursion}, and type analysis proceeds with special considerations. When the type checker is done analyzing a function, its signature is removed from the function call stack.

There is in fact, nothing too special about propagating through a function body in the \textit{recursion} mode. In this mode, propagation executes as normal; however, when the analysis encounters any function call expression whose signature is already present in the function call stack (including that of the current function), the analysis will not enter the encountered function body. Rather, the type checker assigns a \textit{void} type to this call expression, and resumes analysis through the rest of the body.

What we aim to achieve through this procedure, is to poll all the other alternate return nodes of the given function, and return any type information we can. By doing this, we also avoid the type checker from infinitely recursing in its analysis. In the event that the type checker could not determine a specific set of concrete types for the recursive call sequence, it returns an \textit{Any} type, and a warning is
3.3. SOLVING THE CONSTRAINTS

In Fig. 3.26 we show a sample state of the function call stack when we face a hypothetical recursive call sequence. At first, `funcA` is invoked with a certain argument type signature. Within its body, a call expression to `funcB` was encountered, and analysis proceeds through the function. Inside `funcB`'s body, a call expression was encountered with the exact same signature as for the first function call. The type checker recognizes that this is a case of recursion, and pushes through in recursion mode. In this mode, call expressions of `funcB` and `funcA` with the given argument type signatures will be rejected. All other expressions are otherwise, normally evaluated. In Fig. 3.26, we see that another function call (`funcD`) was evaluated from inside the body of `funcA` while in recursion mode. When the recursion mode is analyzed and all available type information gathered, the type checker backtracks to the next call context on the stack, and resumes normal analysis.

The approach we've taken to handle recursive calls seems to work well for sufficiently complex situations. However, we haven't tested this approach extensively. Future work could involve an enhanced iterative approach, where recursive call sequences can be iterated upon until the type information doesn't change for any function involved in the recursion. However, the examples we've considered so far do not seem to need that.
3.3.9 Handling include statements

In chapter 1 we reviewed that the *include* statement essentially copies all contents from an external file into the current file in PHP. The contents may be any PHP code, or HTML markup elements. This is also one way that the analysis could be implemented. In effect, this would involve injecting the entire syntax tree and the corresponding control flow graph in the current analysis.

In our analysis, the ASTs and the corresponding CFG for any included files are created separately. The include and require statements do not cause any additional constraints to be generated. When an include statement is encountered, the type analysis switches over to the script being included. Before this switch, we take the exit type environment from the node preceding the *include* statement in the first (including) script, and then use this type environment as the entry environment for the first node in the second (included) script. This way, the type information for all the identifiers in the first script is available in the second script. This models actual PHP runtime behavior.

When the end of the second script is reached, the exit type environment from the last node is taken and merged with the entry environment of the node, succeeding the *include* statement in the first script. This merging is done using the unification (\( \cup \)) operator. The result of this is that the type information for all identifiers resulting from executing the content of the second script is now also available to the first script. This too is in accordance with the actual PHP runtime behavior.

To illustrate this, let us consider an example. Let ‘fileA’ be defined as in listing 3.27. This file includes a file: ‘fileB’ in the \( k \)th statement.

Let \( \text{exit}_{\text{name}}^i \) and \( \text{entry}_{\text{name}}^i \) represent the exit and entry type environments respectively of node \( i \) in file \( \text{name} \). Let us examine how the *include* statement is handled when the analysis propagates through file *fileA.php*. When the \( k \)th statement is reached, then the analysis switches to file *fileB.php*. At the same time, the following is accomplished:

\[
\text{entry}_{\text{fileB}}^1 = \text{exit}_{\text{fileA}}^{k-1} 
\]

The above sets the entry environment of the first statement in *fileB* as the type environment flowing out of the \( k - 1 \)th statement in *fileA*. The analysis will continue through the body of the script in *fileB*. When the analysis is done in the second file, the exit environment flowing out of this script is merged with the entry type environment of the \( k + 1 \)th statement in *fileA*. I.e.
```php
<?php
    ....
    ...
    [statement]_{k-1};
    include(fileB.php);
    [statement]_{k+1};
    ....
    ...
?>

Figure 3.27 fileA.php

<?php
    [statement]_1;
    ....
    ...
    ...
    ...
    ....
    [statement]_N;
?>

Figure 3.28 fileB.php
The analysis now proceeds normally with the rest of the statements in fileA.

We mentioned in chapter 1 that file inclusion can also be achieved dynamically in PHP. That could happen, for instance, by having a variable in the include statement. This tool, at this stage, is incapable of proceeding in such a case. The reason behind that is that the tool only tracks types of variables (and any expression, in general) during the course of the analysis. Without having specific content information, it is not possible to infer which files to analyze during such a case. Some lateral work may be needed to accomplish this.

### 3.4 Type Analysis On Object Types

In this section, we will review in detail how we model class types in our analysis. At first, we introduce new components that help us in tracking object types through the analysis. This is followed by discussions of the constraints generated by class related operations, and the approach we’ve taken to solve these constraints.

#### 3.4.1 Introducing the Handle type

An object type is represented by the constructor function `Object` for the ADT `Type` in the type checker source. A brief discussion on this type was presented in discussions surrounding Fig. 3.2. We review this type in more detail in this section.

To recapitulate, the `Object` constructor function defined for `Type` is as follows,

```
public data Type = ... Object(Handle h) ...
```

The `Object` type references a parameter: `h`. This parameter is of ADT: `Handle` and provides us a way of introducing a referencing mechanism in our analysis.

A short discussion on object aliasing in PHP was presented in section 1.5.2. In this light, the reason for
using the handle type in our own analysis is quite straightforward. The Handle type in our analysis seeks to emulate the reference type in the PHP engine. When an object instantiation expression is encountered (Fig. 3.31), a new handle is created. This handle is uniquely identified by an integer parameter: id. This handle type is then mapped to the unique object instance via an internal global map variable. When any program variable (in the PHP script) is assigned this object, the type checker assigns a Object type with the newly created id to this variable. When different program variables (in the PHP script) reference this object, the same handle is given to each of aliasing program variable. The steps taken in response to an object instantiation expression are enumerated in Fig. 3.29.

1. Create a new object in the analysis

   \[
   \text{ClassInstance } newObject = \text{objInstance}(id, \text{fieldToTypeMap})
   \]

2. Create a new handle:

   \[
   \text{Handle } newHandle = \text{handle}(id)
   \]

3. Add (handle, object) pair to the global map.

   \[
   \text{HandleToClassInstanceMap} + = (newHandle : newObject)
   \]

4. Return an object type with reference to the handle.

   \[
   \text{return typeSet\{Object(newHandle)\}}
   \]

**Figure 3.29** Instantiating a new object type in the analysis

In Fig. 3.29, we also introduced a new ADT: ClassInstance which is defined as follows:

\[
\text{public data ClassInstance =objInstance(\text{int id, map[Identifier, Type] fieldToTypeMap})|}
\text{statInstance(\text{int id, map[Identifier, Type] fieldToTypeMap})}
\]

This ADT essentially describes a data structure to define object types in our analysis. A class instance, i.e. an object, is represented by a ClassInstance data type instantiated using the objInstance constructor. The role of the second constructor, statInstance is described in the next section.
3.4. TYPE ANALYSIS ON OBJECT TYPES

Aliasing is thus, achieved by passing the handle \(h\) to any aliased variables for this specific object in the program. In this way, we achieve the same kind of aliasing feature supported by PHP 5.

3.4.2 Processing a class definition

A PHP class definition is treated as a top-level statement in our analysis, and it will not generate any constraints. When a new class definition statement is encountered, the analysis will proceed (as in with function definitions) to create a new analysis environment. In the case of a class, this phase involves sorting through each class property, initializing their types to any initialization value, and defining new analysis environments for any methods present in the class.

The most important task in this phase involves creating an internal template representation of the class. New objects of this class will be based off of this representation. This template is stored internally also as a \textit{ClassInstance} algebraic data-type, instantiated using the \textit{statInstance} constructor function. A unique handle is created and issued to this instance. When accessing the types of any static property or method calls in the program, this static instance is invoked. However, unlike with a regular class instance, the handle to a static class instance never gets assigned to a program variable. The steps that are taken when a new class definition is encountered are enumerated in Fig. 3.30.

1. Create a new static-instance type in the analysis

\[
\text{ClassInstance } \text{newClass} = \text{statInstance}(id, \text{staticFieldToTypeMap})
\]

2. Create a new handle:

\[
\text{Handle } \text{newHandle} = \text{handle}(id)
\]

3. Map handle to static instance.

\[
\text{HandleToClassInstanceMap} += (\text{newHandle} : \text{statInstance})
\]

\textbf{Figure 3.30} Processing a class definition in the analysis.

The static instance maintains the repository of type information for all the static class properties and methods. This ensures that individual copies of the static properties are not distributed among
object instances spawned during the course of the type analysis. This instance is also used to provide context for analyzing any static method invocations later in the analysis. We will review this in more details as additional discussions are provided for typing method calls.

3.4.3 Constraints for class-related expressions

In order to handle expressions involving class-related operations, we have to define new kinds of constraints. The constraints listed Fig. 3.31 deal exclusively with class-related operations. In particular, they handle expressions involving the creation a new object instance, access to an object or class property, and invocation of a class or instance method.

\[
C^l\left[\left(\text{new } c\text{Name}\left(e_1^{l_1}, \ldots, e_n^{l_n}\right)\right)^l\right] = \{ l := \text{newObjectInstance}\left(c\text{Name}, [l_1, \ldots, l_n]\right)\}
\]

\[
C^l\left[\left(v_0^{l_0} = e_0^{l_0} \rightarrow \text{var}\right)^l\right] = \{ l_1 := \text{fromObjectProperty}(l_0, \text{var})\}
\]

\[
\bigcup \{ l_0 \equiv \text{object}(\text{any}), l_2 \leftarrow l_1\}
\]

\[
C^l\left[\left(v_0^{l_0+2} = e_0^{l_0} \rightarrow f\text{name}\left(e_1^{l_1}, \ldots, e_n^{l_n}\right)^{l_{n+1}}\right)^l\right] = \{ l_{n+1} := \text{fromMethodCall}(l_0, \text{name}_f, [l_1, \ldots, l_n])\}
\]

\[
\bigcup \{ l_0 \equiv \text{object}(\text{any}), l_{n+2} \leftarrow l_{n+1}\}
\]

\[
C^l\left[\left(v_0^{l_0} = c\text{Name}:: \text{var}\right)^l_2\right] = \{ l_1 \leftarrow l_2, l_2 := \text{fromStaticProperty}(c\text{Name}, \text{var})\}
\]

\[
C^l\left[\left(v_0^{l_0+2} = c\text{Name}:: f\text{name}\left(e_1^{l_1}, \ldots, e_n^{l_n}\right)^{l_{n+1}}\right)^l\right] = \{ l_{n+1} := \text{fromStaticCall}(c\text{Name}, \text{name}_f, [l_1, \ldots, l_n])\}
\]

\[
\bigcup \{ l_{n+2} \leftarrow l_{n+1}\}
\]

Figure 3.31 Constraints generated by class operations.

In order to process the constraints in Fig. 3.31 we’ve retrofitted the get_concrete_type_of rascal function presented in Fig. 3.22 to handle the new cases. The updated algorithm is presented in Fig. 3.32. We briefly review the steps taken in response to each constraint processed by the updated algorithm.
3.4.3.1 Create new objects

Earlier in section 3.4.1, we discussed how the expression for creating a new object instance is handled. In summary, the constraints for creating a new object instance returns an object type with a handle argument. A unique Rascal data of the ClassInstance data type is created to represent the PHP object. An internal global map data (HandleToClassInstanceMap) stores this handle, class instance pair as a key, value pair. Any subsequent reference to the handle is then mapped to the corresponding class instance data.

3.4.3.2 Retrieving class property types

Retrieving instance and class properties is a fairly straightforward task. An initial lookup is performed to retrieve the handle(s) to look up a particular class instance data. If the expression in context does not hold an Object type, then a warning is issued and a void type is used. When the handle(s) is available, the global map HandleToClassInstance is queried to retrieve the specific class instance data associated with the handle. The fieldToTypeMap argument of the retrieved class instance is used to retrieve the type of the given class property.

If the given class property is not found in the fieldToTypeMap argument, a warning is issued with message specifying the nature and location of the error. Similarly, if the definition for the class with name: cName is not found, or the expression specified by the label: l_0 is not a valid class instance, a warning with information on the nature and location of the error is issued in each situation.

3.4.3.3 Handling method calls

A method is a function that appears within a class body. When a method call is made, it is necessary to keep track of the context in which the call was made. The context for a method call can be understood as the particular object or class that invoked the function. Method calls can be made by objects outside of the class body, or within the body of one of its method. These instance methods may use the $this identifier to access any of the calling object's instance property, and they may use the self identifier to access any of the static (Class) property. A method body may also contain code to instantiate new objects of any class type, and call methods on those class instances. In this section, we discuss the approach we took to resolve these method call contexts.
function GET_CONCRETE_TYPE_OF($l_k$, $constraints$, $entry$)
  $ab := cc :\leftarrow where cc \in constraints$, and $cc.l = l_k$;
  switch $ab$ do
      .........
      ....
    case newObjectInstance($cName$):
        return type_set({Object($h$)}); where $h = new$ handle($id$)
        & HandleToClassInstanceMap + = [h \mapsto new$ objInstance($id$, fieldToTypeMap)]
        & fieldToTypeMap $\leftarrow$ GlobalClassDeclMap[$cName$].initializedMap
    case fromObjectProperty($l_0$, var):
        return map[var]; where map $\leftarrow$ objInstance.fieldToTypeMap
        & objInstance $\leftarrow$ HandleToClassInstanceMap[objectType.$h$]
        & objectType $\leftarrow$ get_concrete_type_of($l_0$, $constraints$, $entry$)
    case fromStaticProperty($cName$, var):
        return map[var]; where map $\leftarrow$ statInstance.fieldToTypeMap
        & statInstance $\leftarrow$ HandleToClassInstanceMap[$h$]
        & $h \leftarrow$ GlobalClassDeclMap[$cName$]
    case fromMethodCall[$l_0$, name, [$l_1$,..., $l_n$]]:
        $\rightarrow$ GlobalObjectContextStack.push(objectType.$h$)
        & objectType $\leftarrow$ get_concrete_type_of($l_0$, $constraints$, $entry$)
        $\rightarrow$ toReturn $=$ type_set
        {analysis$_f$([[ti | ti $\leftarrow$ get_concrete_type_of($l_i$, $constraints$, $entry$)])]}
        $\rightarrow$ GlobalObjectContextStack.pop(objectType.$h$)
        $\rightarrow$ return toReturn
    case fromStaticCall[$cName$, name, [$l_1$,..., $l_n$]]:
        $\rightarrow$ GlobalObjectContextStack.push($h$)
        & $h \leftarrow$ GlobalClassDeclMap[$cName$]
        $\rightarrow$ toReturn $=$ type_set
        {analysis$_f$([[ti | ti $\leftarrow$ get_concrete_type_of($l_i$, $constraints$, $entry$)])]}
        $\rightarrow$ GlobalObjectContextStack.pop($h$)
        $\rightarrow$ return toReturn
    case default:
        return $ab$;
  end function

Figure 3.32 The updated get concrete type root function.
3.4. TYPE ANALYSIS ON OBJECT TYPES

An instance method call expression generates the constraints: \( \text{fromMethodCall}(l_0, name_f, [l_1, \ldots, l_n]) \). The label \( l_0 \) references the object expression on which the method is invoked. Likewise, a static method call gives rise to the constraint: \( \text{fromStaticCall}(cName, name_f, [l_1, \ldots, l_n]) \). In either case, the first step taken is to identify the set of handles that corresponds to the class instance that invoked the method call, \( name_f \). This set of handles is pushed onto a global stack structure: \( \text{GlobalObjectContextStack} \) that accepts the corresponding data type for argument. Secondly, the concrete types of each of the argument in the method is inferred. Then the analysis is handed to the section that manages type analysis for function calls (section 3.3.7) along with the inferred types of the arguments.

With the information available in the global stack data: \( \text{GlobalObjectContextStack} \), it is now possible to resolve the context of the method call. Before the type analysis propagates through the method body, the variables identified with \$this\ is initialized to an object type of the handles that sits at the top of the stack. Any reference to \$this\ in the method is subsequently mapped to the object whose method was invoked. When analysis through the method body is completed, the set of handle on the top of the stack is popped off.

Note that we've repeatedly identified a method call context to a set of handles rather than one handle. The reason for that is to account for branching statements that may associate multiple instances to a variable. Consider the script in Fig. 3.33.

```php
if(…) { 
    $var = new A();
    ...
} 
else { 
    $var = new B();
    ...
    ...
} 
$var->someMethodCall(…)
```

Figure 3.33 Program variable has two distinct object types
In Fig. 3.33, $var$ would be associated with two handles, each one of which points to a unique object instance created when the particular branch is analyzed. While processing the method invocation statement, it is thus necessary to map $var$ with both the handles that are associated with $var$. There maybe a better strategy to handle this situation without storing both the handles. However, we do not do that currently and it maybe a recommended improvement for future work.

### 3.4.3.4 Combining object types

The last discussion we will provide on this chapter is on how we handle situations where object types need to be combined. Because of the dynamic nature of PHP, branching statements may introduce multiple object types on the same program variable. We saw an example of that in Fig. 3.33. There are a couple of different scenarios we treat individually, and we will outline them in the rest of this section.

- **Object types with same handles:**
  When the alternate branches reference the same object created before the split happened, then we simply combine the types of any of the property that may have been initialized differently on the alternate branches. The types of the properties are combined using the $\triangledown$ operator defined in Fig. 3.5. The program variable $varA$ in Fig. 3.34 gives an example of this case.

- **Object types with same parent class:**
  When the alternate branches each includes a statement that creates a new class instance of the same class for the same program variable, we combine them into one. For example, in Fig. 3.34, each of the two branches create a unique object instance of class B, and assign that to variable $varB$. Each object instantiation expression also issues unique handle ($h_1$ and $h_2$ in this case). However, when we merge these types, we maintain reference to only one of the handle, and combine the types of the class property available from all the alternate branches using the standard $\triangledown$ operator.

- **Object types instantiated from different parent classes:**
  Finally, we have a situation where the alternate branches include statements that assign different class types to a program variable. An example can be referenced from Fig. 3.33. In such a scenario, we need to associate both the object types with the given identifier. For example, in Fig. 3.33, the resulting type of $var$ at the end of the branching statement would
3.5. CONCLUSION

be as follows:

\[ \text{typeOf}($\text{var}) = \{ \text{Object}(h_i), \text{Object}(h_j) \} \]

where \( h_i \) and \( h_j \) represent the handle ids that were issued when the object instantiation expressions were encountered in the branches.

```php
<?php
$varA = new A();     // handle: h1 issued
...
if ($cond) {
    $varB = new B();   // handle: h2 issued
    $varB->field = "String";  // assign string type
    ...
    $varA->field = true;  // assign bool type
} else {
    $varB = new B();   // handle: h3 issued
    $varB->field = 13;  // assign int type
    ...
    $varA->field = 23.1; // assign float type
}

// typeOf($varA) = \{ Object(h1) \}, typeOf($varA->$field) = \{ bool, float \}
echo $varA->field;

// typeOf($varB) = \{ Object(h2) \}, typeOf($varB->$field) = \{ string, int \}
echo $varB->field;
```

Figure 3.34 Program variable has two distinct object types

### 3.5 Conclusion

In this chapter, we examined the details of the components that constitute our analysis. We saw that we leverage the capability of the PHP-Air [HK14] framework to generate a control flow graph and AST of the target PHP code, and directly work on these data structures to implement the type analysis. The analysis itself is fundamentally based on the fixed point iteration methodology, and utilizes a
custom widening operator for stability and conciseness. Another critical constituent of the analysis is a system of regularized constraints that is solved to infer types on all the relevant expressions of the target PHP script. Although our analysis is far from being able to type every possible case in PHP, a strong foundation of type checking features is currently set in place.
In this section, we guide the readers through many scenarios of the application of the type analyzer developed in this work. Also, to shorten future references, we shall call it PType. We provide example PHP scripts where possible typing errors have been introduced in the script. Because PHP is a dynamically typed scripting language, the PHP compiler will not detect these errors before actual execution. Many times the PHP compiler will fail with a fatal runtime error, which could lead to catastrophic loss of computational data and time. In some cases, the PHP compiler will apply some internal type coercion rules to read data of inconsistent types.

While some of this conversion may not introduce an error, at other times, this could lead to an unexpected bug that may be very hard to find. We believe that all of these scenarios are detrimental to the overall quality of the code-base, resulting in applications that are harder to maintain and debug. PType aims to catch these errors without actually running the PHP application. The caught errors are displayed as specific warning messages. We demonstrate samples of the kinds of errors that PType can catch in the rest of this chapter.
4.1 Errors in Simple Expressions

4.1.1 Type coercion errors

The first class of errors we demonstrate is related to simple kinds of type-coercions. In Fig. 4.1, we begin by defining four variables, each initialized to a unique kind of data. In the rest of the script, each of these variables is subject to a variety of numerical, boolean, and string operations. PType must be able to inform the developer of any incompatible operations being applied on a value. Fig. 4.2 presents the labelled warnings that were generated by running PType in the mentioned PHP script.

The first two warnings (i.e. 1 and 2) in Fig. 4.2 corresponds to the statements that involve evaluating the negative values of data in variables: $a, and $b, and assigning them to $test. Taking the negative values of bool and string types do not make sense, and more often than not, these expressions represent a logical error on the part of the user. Warnings 1. and 2. promptly bring attention to these issues in the given code.

PHP itself ignored these issues and continued execution through the script. The PHP compiler returned -1 when evaluating the negative value of $a, and it returned 0 when evaluating the negative value of $b. This is a result of PHP’s internal arithmetic type-casting rules. According to these rules, the true boolean value assumes a numerical value of 1, and hence the negative of $a returned -1. Likewise, a string data is internally coerced to a value of 0. Thus, the negative of $b also returned 0.

In our analysis, we assign the num type to $test in the above cases (lines: 8-9). We do so because evaluating the negative of an expression leads us to conclude that a num type will result at the end of the evaluation.

Warnings 3.-5. in Fig. 4.2 represent type coercions errors that resulted from lines 11 and 12 of Fig. 4.1. The increment operator specifically expects integer arguments. These warnings were generated as a result of applying the increment operator on boolean, string, and float data respectively. The warnings also provide brief explanations and the particular location of the errors.

Warnings 7. through 9., are generated by attempting to use the boolean negation operator on the given list of program variables. These warnings are generated because the script attempts to treat
<?php
$a = true;
$b = "Hello";
$c = 4.53;
$d = 99;

// try to change the sign
$test = -$a;  $test = -$b;
$test = -$c;  $test = -$d;

// try to use the increment operator
$test = $a++;  $test = $b++;  $test = $c++;  $test = $d++;  

// try to use data as boolean expression
$test = !($a);  $test = !($b);
$test = !($c);  $test = !($d);  

// apply some binary numerical operations
$test = 99 - $c;
$test = $c + $d;  
$test = $a  $b;
$test = $b / $d;  
$test = $c % $a;
$foo = (1.23+ $a)  $c / ($b - $d) % $b;
?>

Figure 4.1 Simple data coercion examples in PHP
4.1. ERRORS IN SIMPLE EXPRESSIONS

CHAPTER 4. EVALUATIONS OF ERRORS

1. typeCoercionWarning("Possible Illegal Type Coercion @ lab(18). Got: Bool. Expected from: Num.")
2. typeCoercionWarning("Possible Illegal Type Coercion @ lab(23). Got: String. Expected from: Num.")
3. typeCoercionWarning("Possible Illegal Type Coercion @ lab(38). Got: Bool. Expected from: Int.")
4. typeCoercionWarning("Possible Illegal Type Coercion @ lab(43). Got: String. Expected from: Int.")
5. typeCoercionWarning("Possible Illegal Type Coercion @ lab(48). Got: Float. Expected from: Int.")
6. typeOfVarChanged(" Var(test) changed type @ lab(57) from Int to Bool.")
7. typeCoercionWarning("Possible Illegal Type Coercion @ lab(63). Got: String. Expected from: Bool.")
8. typeCoercionWarning("Possible Illegal Type Coercion @ lab(68). Got: Float. Expected from: Bool.")
9. typeCoercionWarning("Possible Illegal Type Coercion @ lab(73). Got: Int. Expected from: Bool.")
10. typeOfVarChanged(" Var(test) changed type @ lab(77) from Bool to Num.")
11. typeCoercionWarning("Possible Illegal Type Coercion @ lab(90). Got: Bool. Expected from: Num.")
12. typeCoercionWarning("Possible Illegal Type Coercion @ lab(91). Got: String. Expected from: Num.")
13. typeCoercionWarning("Possible Illegal Type Coercion @ lab(96). Got: String. Expected from: Num.")
14. typeCoercionWarning("Possible Illegal Type Coercion @ lab(103). Got: Bool. Expected from: Num.")
15. typeCoercionWarning("Possible Illegal Type Coercion @ lab(117). Got: String. Expected from: Num.")
16. typeCoercionWarning("Possible Illegal Type Coercion @ lab(113). Got: String. Expected from: Num.")
17. typeCoercionWarning("Possible Illegal Type Coercion @ lab(109). Got: Bool. Expected from: Num.")

Figure 4.2 Simple data coercion examples in PHP
non-boolean data as boolean expressions. For non-boolean data, the PHP compiler usually type-casts the non-boolean expression into a boolean value. In general, empty values such as empty strings, empty arrays, and unset variables (including the numeric value of 0) are treated as false expressions. Everything else, including resources, is treated as a true expression. The complete specifications of PHP’s boolean type-casting mechanism can be found in the online PHP manual for type juggling [Phpa].

Finally, warnings 11 through 17 in Fig. 4.2 specify scenarios in which non-numerical expressions are used in binary numerical operators. They correspond to statements in lines 20 - 27 in Fig. 4.1. These expressions are clearly erroneous and caused the PHP compiler to throw a fatal runtime error.

In the cases mentioned in this section, the PHP compiler does its best to cast values to the right type. While this may alleviate the need for manual type-castings, they can be sources of hard-to-find bugs in an application. PType identifies such expressions, and issues warnings with information on the location and nature of the error.

4.1.2 Array type-coercions

In PHP, an array is essentially an ordered map of key-value pairs. The key can be a string or an int. The values can be of mixed types. Key-value pairs can be added on the fly, or be removed by using the unset function. Arrays can be multidimensional, and can contain data structures such as trees and arrays. Because of the vast number of ways in which data in/from arrays can be manipulated, users are prone to making errors with array types. PType supports certain array type checking features that could be quite useful. Some scenarios are given in this section.

In the script in Fig. 4.3, we have created two arrays: one a strictly string array ($strArr), and the other a strictly integer array ($numArr). We also have a simple variable that holds a string data ($nonArr). This is followed by expressions that manipulate the array variables in different ways. We have introduced various kinds of errors in the script, and we briefly review them next.

In line 9 of Fig. 4.3 we have attempted to apply the count function to a non-array. This results in the first warning, since we’ve supplied a string type where an array type is expected. In line 20, we get the second warning as a result of retrieving values for the string array, and subjecting them to the numerical addition (+) operator. In the next statement, (line 24), we get two warnings: one as a result of using the string concatenation (. or dot ) operator on a numeric value, and the other
because the type of variable: $test changed from num to a string type. The second happens so
because through statements in line 20 and line 24, we conclude that the program var $test assumes
the num and string types respectively. These conclusions are made on the basis of the numerical
and string operator that each appears respectively in lines 20 and 24.

Line 36 generates additional warnings in the script. This is motivated by the fact that $strArr now
holds more than one type of values. We believe that it is generally a risk-prone practice to allow
arrays to hold data of more than one kind. Hence, we inform the user that $strArr acquired more
than one type. This also triggers the warning that the type of the given var changed. Now, because
we added a numeric type to the $strArr program variable, line 36 generates a warning in response
to using the string dot operator on a variable that could potentially return a numeric type. In the
final line, line 41, we add a key-value pair to the simple program var, $nonArr, effectively rendering
it as an array. This triggers the warning that such an operation is being performed, since it is an
unconventional operation leading to surreptitious program bugs.

1. typeCoercionWarning("Possible Illegal Type Coercion @ lab(28). Got: String(). Expected from: Ar-
array(Any()).")
2. typeCoercionWarning("Possible Illegal Type Coercion @ lab(44). Got: String(). Expected from: Num().")
3. typeOfVarChanged(" Var(test) changed type @ lab(48) from Num() to String().")
4. typeCoercionWarning("Possible Illegal Type Coercion @ lab(50). Got: Array(Int()). Expected from:
String().")
5. noWarning("Processed constraint to add type to Array: var(strArr) @ lab(62). ")
6. typeOfVarChanged(" Var(strArr) changed type @ lab(62) from Array(String()) to Ar-
array(String()),Array(Float()).")
7. varHasMultipleTypes("Var strArr acquired more than one types at lab(66).")
8. typeCoercionWarning("Possible Illegal Type Coercion @ lab(72). Got: Float(). Expected from: String().")
9. typeCoercionWarning("Trying to add key -> value pair to non-Array: var(nonArr) @ lab(76).")

Figure 4.4 Simple data coercion examples in PHP

We saw here that PHP arrays expose a variety of avenues through which errors can be introduced in
<!-- start of page -->

4.1. ERRORS IN SIMPLE EXPRESSIONS

CHAPTER 4. EVALUATIONS OF ERRORS

```php
<?php
$strArr = array(10 => "a", 20 => "b", 30 => "c");

$numArr = array(1, 2, 3);

$nonArr = "A\_String";

@test = count($strArr);  // Warnings: [1]
@test = count($nonArr);  //

//subject elements of arrays to binary operations

//legal, addition
//valid for numbers
@test = 2 + $numArr[1];

//illegal, addition
//not defined for strings
@test = 2 + $strArr[10];  // Warnings: [2]

//illegal, string
//concatenate on numbers
@test = "abcd" . $numArr;  // Warnings: [3, 4]

//legal, string concatenate defined for strings
@test = "abcd" . $strArr[20];

//changing array type by
//adding a new type
$strArr[40] = 1.24;  // Warnings: [5, 6, 7]

// must throw a flag, because one
//of the elements could be a num
@test = "abc" . $strArr[20];  // Warnings: [8]

//Now, change var to an array
//generate warning: bad practice!
?>
```

---

**Figure 4.3** Type coercion errors from array types

---

86
the program. The error catching features of PType are primarily centered around the fact that a PHP array must hold a unique type of values, and that mixing values, while convenient, may introduce hard-to-find bugs in the application. PType is able to catch these errors, and many more as are presented in Fig. 4.4.

4.1.3 Miscellaneous errors

In Fig. 4.5 we demonstrate a few other cases which causes PType to generate additional kinds of warnings. These cases are associated with expressions that normally return a null type during runtime. These kinds of warnings are subtle, easily overlooked, and have the potential to cause fatal runtime errors. Hence, we consider these kinds of errors as high-priority bugs.

The first two warnings in Fig. 4.6 are generated in response to calling a function that returns a void type, and making that assignment to the program var $c. In line 16, we attempt to invoke an undefined function: $callUnDeclaredFunc, that results in a warning (#3) that states that a function definition by that name was not found. Warning #4 informs that a void type was subsequently used for the expression, and warning #5 alerts that the var $d was assigned the void type.

In line 12 of Fig. 4.5, we attempt to use a variable which has not been declared previously. Consequently, the type checker informs the user that the variable has not been initialized (warning #6). As in before, the type analysis uses the void type for $var and pushes on with the analysis.

We show a more interesting type analysis case in line 16 of the given PHP script (Fig. 4.5). This statement involves calling the callDeclaredFunc function with an integer argument. The function definition itself attempts to use the parameter as an array type in its body. When the function call expression in line 16 is evaluated, analysis propagates through the callDeclaredFunc body. This generates a (quite) verbose set of warnings because an integer type has been supplied where array types are expected. When PType encounters an expression involving a subscription operator ([key]) and an array type is not found, it will issue a warning and use a void type for that expression. Warnings 11-22 are essentially a manifestation of this type checking feature. These warnings report instances where the int type was found where an array type was expected. Furthermore, in each of this instant, warnings are generated to inform that a void type was used for the expression, and assigned to a program variable when applicable.

Finally the analysis exits from the callDeclaredFunc body, and back to the main script. Since a void
type was essentially returned from the function analysis (line 32, Fig. 4.5), additional warnings: #9 & #10, are issued to alert that more expressions have resulted in *void* types in the main script.
4.1. ERRORS IN SIMPLE EXPRESSIONS

```php
<?php
// define a simple var
$b = 12;

// call a declared func
$c = callVoidFunc($b); Warnings: [1, 2]

// call an undeclared function
$d = callUnDeclaredFunc($b); Warnings: [3, 4, 5]

// access uninitialized var
$e = (45 + $var) - 12; Warnings: [6, 7, 8]

// call declared func with non-suited argument
// access uninitialized var
$g = callDeclaredFunc($b); Warnings: [9, 10]

//------------------- function definitions-------------------
function callVoidFunc($x)
{
    // do something useful
    $temp = $x - 1;
    $perm = ($temp) % 3;
    return;
}
function callDeclaredFunc($x)
{
    // expects an array type
    if (!($x[0] < 5)) Warnings: [11–14]
        $x[0] = $x[0] + 5;
        return $x[0]; Warnings: [15–17]
}
?>
```

Figure 4.5 Other miscellaneous warnings
4.1. ERRORS IN SIMPLE EXPRESSIONS

CHAPTER 4. EVALUATIONS OF ERRORS

1. usingVoidTypeForExpr("Computed typeSet for lab(5) has Void type.")
2. assignVoidTypeToVar("Assigning a Void type to var(c) @ lab(5).")
3. failedToFindFunction("Failed to find function definition: callUnDeclaredFunc, call site @ lab(13).")
4. usingVoidTypeForExpr("Computed typeSet for lab(13) has Void type.")
5. assignVoidTypeToVar("Assigning a Void type to var(d) @ lab(10).")
6. readFromNonDeclaredVar("Attempt to read from undeclared var(var) @ lab(17).")
7. usingVoidTypeForExpr("Computed typeSet for lab(17) has Void type.")
8. typeCoercionWarning("Possible Illegal Type Coercion @ lab(17). Got: Void(). Expected from: Num().")
9. usingVoidTypeForExpr("Computed typeSet for lab(26) has Void type.")
10. assignVoidTypeToVar("Assigning a Void type to var(g) @ lab(23).")
11. readFromNonArrayVal("Expected array @ lab(42). Got Non-array: typeSet(Int()).")
12. usingVoidTypeForExpr("Computed typeSet for lab(44) has Void type.")
13. typeCoercionWarning("Possible Illegal Type Coercion @ lab(44). Got: Void(). Expected from: Num().")
14. typeCoercionWarning("Possible Illegal Type Coercion @ lab(42). Got: Int(). Expected from: Array(Any()).")
15. readFromNonArrayVal("Expected array @ lab(59). Got Non-array: typeSet(Int()).")
16. usingVoidTypeForExpr("Computed typeSet for lab(62) has Void type.")
17. typeCoercionWarning("Possible Illegal Type Coercion @ lab(59). Got: Int(). Expected from: Array(Any()).")
18. readFromNonArrayVal("Expected array @ lab(51). Got Non-array: typeSet(Int()).")
19. usingVoidTypeForExpr("Computed typeSet for lab(53) has Void type.")
20. typeCoercionWarning("Trying to add key -> value pair to non-Array: var(x) @ lab(48).")
21. typeCoercionWarning("Possible Illegal Type Coercion @ lab(51). Got: Int(). Expected from: Array(Any()).")
22. typeCoercionWarning("Possible Illegal Type Coercion @ lab(53). Got: Void(). Expected from: Num().")

Figure 4.6 Warnings generated for the main script in Fig. 4.5
4.2 Errors in Function Calls

PHP functions are quite dynamic in nature. They do not have a specified return type. They can accept an arbitrary number of arguments, as well as be defined as members of classes. Because of these dynamic natures, PHP functions can lead to many type related errors in the program. In this section, we review some of the function-related type checking features that we included in PType.

In section 4.1.3 we reviewed a scenario in which we attempted to call a function whose definition was not available. We also included scenarios where a function call that returned a null type was assigned to a program variable, and where an incorrect type was passed an argument to a function call expression. In this section we review additional scenarios that result in warnings to be generated.

Fig. 4.7 lists the function someFunction that can change the state of an object argument. Somewhere in the function, the function sets the property $var of object instance $x to a string data. This action may or not be intended, but if the object being passed in the function call already defines a property with the same name, then the data in this property will be replaced. This replacement is permanent, so that when execution resumes on the calling script, then the data on this property is of the new type.

```php

// a func. that mutates
function someFunction($x) {
    $x->var="hello!";
}

// call mutates obj state
$Result=someFuncCall($obj);
```

Figure 4.7 Warnings related to function calls
When an instance such as in Fig. 4.5 is encountered, PType emits a warning that informs the user that such an operation is happening. A new type of warning is generated in this case, pointing to the nature and location of the source of the location. An example of such a warning is given in Fig. 4.8

- ...  
- objChangedState("Object instance: var(obj) changed state in a func. or method body @ lab(19).")

**Figure 4.8** Warning generated to inform of an object state change

Next, we present an erroneous situation involving recursive function calls. It maybe the case that a recursive function call sequence lacks a base expression, in which case the recursion may continue until there is a memory exhaustion. In such situations, PType detects such a case and reports it as a new type of warning. This is demonstrated in Fig. 4.9.

```
// a func. that recurses infinitely  
function recursiveFunc($val) {
    // do important things
    ...... 
    .... 
    // forget to add base case 
    return recursiveFunc($val);
}

......
$result=recursiveFunc($data);
```

**Figure 4.9** Warnings related to function calls

Calling `recursiveFunc` as defined in Fig. 4.9 will end up with a warning listed in Fig. 4.10. It is to
be noted that PType is also able to analyze recursive call sequences involving multiple function definitions. In general, if PType detects a potential recursive call sequence, and is unable to locate base cases that support a proper recursion rollback, warnings as in Fig. 4.10 are generated to inform the user about this situation.

**Figure 4.10** Warning generated by a recursive function call sequence without a base type

And lastly, we review one additional kind of warning associated with function calls before we end this section. This kind of warning is associated with a function call expression where the number of supplied arguments do not match the number of formal parameters in the function definition (Fig. 4.11). The PHP compiler continues executing in such instances, while emitting warnings about the null arguments. Following suit, PType also emits a warning in response to such an event, and proceeds with analyzing the function body. The parameters corresponding to the missing arguments are each assigned a `void` type and used as such through the function analysis. Expressions that read from these variables will spur additional warnings as a result of using the `void` values.

```php
<?php
function func($x, $y, $z)
{
    return ($x $y) %$z;
}
$result = func(1, 2);
?>
```

throws the following warning at the function call site,

**Figure 4.11** Calling a function with insufficient arguments.

```php
<?php
function func($x, $y, $z)
{
    return ($x $y) %$z;
}
$result = func(1, 2);
?>
```

```php
throws the following warning at the function call site,
```
In this section, we reviewed a number of different error catching features that we added support to in PType. We saw that the dynamic nature of PHP functions provide opportunities to introduce many different kinds of errors. These features allow PType to catch many frequently occurring errors related to function definitions and function call expressions. In the next few sections, we list features to catch errors associated with class and object related expressions. Such expressions allow for an even wider array of opportunities to introduce subtle bugs in a PHP application.

4.3 Errors in Class Expressions

Adding type analysis for classes and objects leads to an explosion of the variety of errors that PType needs to grapple with. In this section, we review the different kinds of class-related typing errors that it is capable of catching. Before we start, we review a basic object-oriented system for reading data from a data stream.

The first class is a representation of a data element. We call this the DataRow class (Fig. 4.12). It has one field, $mycontent, and exposes a getter and setter method to get and set the $mycontent property respectively. The second classDataStream (Fig. 4.12) manages the resources to a data stream. It is based on the singleton pattern model and exposes only a single instance of the resource stream object, represented by the $myResource class property.

Finally, we have a third class called DataReader (Fig. 4.13) that is designed to read a data row item. This class exposes the method: getRandomData, that can be used to get a new instance of the DataRow class. Of course, the current implementation of this method is very basic and returns identical data-row items with each invocation. However, our goal here is to present type checking features and we do not focus on the implementation specifics of any method.

Now that we have our classes, we call the statements in Fig. 4.14 to create a sample instance of our data reader framework. Running PType through this system generates a few low priority warnings. In particular, in the main script, we get a warning stating that the $stream object undergoes a state change while calling a method (Warning #1, Fig. 4.15). This warning was generated in response to the fact that during the method call setMyResource, the state of the $stream object is changed. Warning #2 (Fig. 4.15) is generated because of the presence of a float data in the PHP echo statement. This triggers the typeCoercionWarning since we've established the criteria that only data of string type may appear in the echo statement.
In order to illustrate the different kinds of error catching features built in PType, we start introducing subtle bugs. Some of these bugs may trigger a fatal error, a warning, or nothing depending on the severity of the error. We provide small discussions on each error and the warnings generated in the next few subsections.

### 4.3.1 Accessing an undefined class property

The *accessUndefinedClassProperty* warning is produced whenever a given class property cannot be found for a property access expression. This applies for both static and instance class properties. This is a common source of errors, and would be useful to have in any type checking tool. The illustration of a scenario involved is given in this section.

The error we introduce is to illustrate this class of error is to change the access identifier for a static class property. Static properties need to be accessed inside the parent class method using the *self* identifier. We've altered that to use the *$this* identifier in this scenario. The affected method is the *setMyResource* method in the *DataStream* class, and is shown in Fig. 4.16.

The warnings in Fig. 4.17 are produced when we've introduced the error in Fig. 4.16 in our data reader system. The warnings themselves are quite easy to understand. The first warning informs the user that the type analyzer could not find a property with the name *resourceSet*. Consequently, PType assigns a *void* type to the expression (which is also emitted as a warning) and continues. Since this expression appears where a *boolean* type is expected, PType additionally emits the third warning stating that this happened. Our goal here is to produce the warnings quite liberally. A different front-end logic could easily be written to sort out possible redundant warnings, or filter out warnings below a certain priority threshold.

The PHP compiler emitted a *E_NOTICE* message when encountering this error. This is an indication that there maybe a possible bug in the specified code location. PType, in this context, is more useful because it will analyze all reachable statements of the given PHP application, and warn us about this bug. The PHP compiler only follows the linear execution of the code and if the conditions are not satisfied such that this statement is executed, a bug such as this could stay hidden for a long time in the application. PType could also be easily be programmed to perform more analysis on these errors. For e.g. It could first attempt to check if a matching static property was found, or if the property was available in a super class. These could be recommended work for future and we do...
4.3. ERRORS IN CLASS EXPRESSIONS

not delve more into these aspects.

4.3.2 Reading from uninitialized class property

A type checking feature that we decided would be quite useful to have is the ability to detect if an expression accesses an uninitialized class property. This is important because it is a commonly occurring scenario and could lead to subtle application bugs. An example of this scenario is presented in Fig. 4.18.

In Fig. 4.18, we've commented out the script that authenticates the $reader object before it is assigned a data stream. The method setDataStream specifically checks to see if the object property $authenticated is true before setting the stream. In this scenario, the $authenticated property is uninitialized, and PHP interprets that as false. We've chosen to mark this situation as a potential bug, and generate a warning to inform the user (Fig. 4.19, Warning #1). While this situation is relatively benign, other situations such as these could lead to null values that throw fatal runtime errors. In a worse case scenario, PHP may use coercion to consume the null value to induce unexpected and intractable bugs.

4.3.3 Calling method on non-object variable

Another error checking feature we've included in our type checker is the ability to infer if a method call or property access operation is invoked on a non-object variable. Consider the script in Fig. 4.20.

The script in Fig. 4.20 initializes the program var $dataObj in two different ways in the conditional branches. At the end of the branching, a method invocation is performed on $dataObj. If the execution had followed the else branch, this would trigger a fatal runtime error causing the program to crash. The type checker correctly diagnoses this scenario, and informs the user that this may happen. This list of warnings produced is given in Fig. 4.21.

Without listing out all the warnings, it is interesting to note that PType also reports that $dataObj (Warning #3, Fig. 4.21) acquired more than one type in the script given in Fig. 4.20. We believe that this scenario can cause unexpected future errors, and must be avoided for a more robust application.
4.3.4 Dynamic insertion of object properties

Unlike strongly typed language like Java, it is possible to dynamically add new instance properties in PHP. A sample of how this maybe achieved is given in Fig. 4.22. Once the property is inserted, it is possible to access the property using only the particular object to which the property is bound to. We believe that this is a vulnerable aspect of PHP and must be issued as a potentially erroneous site.

Instigating the change in Fig. 4.22 causes a couple of warnings to be generated while type-analyzing the script. More prominent than others are the warnings that specify that a dynamic field insertion event occurred, and that an uninitialized class property was accessed. These warnings are shown in Fig. 4.23. It is actually the case that excluding the character $d from the property-set expression $reader authenticate results in a new class property (authenticate) to be bound to the object $reader, and the original property by name authenticated remains uninitialized. The PHP compiler itself will not emit any warning or notice and continue execution as usual. PType successfully flags this situation for recommended review.
class DataRow{
    private $mycontent;
    //______________________________
    public function setMyContent($str){
        $this->mycontent=$str;
    }
    //______________________________
    public function getMyContent(){
        return $this->mycontent;
    }
}

class DataStream{
    private $myResource;
    private static $resourceSet=false;
    //______________________________
    public function setMyResource($val){
        if (!$resourceSet){
            $this->myResource=$val;
            self::$resourceSet=true;
        }
        return $this;
    }
    //______________________________
    public static function isResourceSet(){
        return self::$resourceSet;
    }
}

Figure 4.12 The DataRow and DataStream class in our sample data reader system.
4.3. ERRORS IN CLASS EXPRESSIONS  

CHAPTER 4. EVALUATIONS OF ERRORS

```php
class DataReader{
    public $authenticated;
    private static $dataStream;
    //______________________________
    public function setDataStream($newDataStream)
    {$cond=$this->authenticated;
    if ($cond){
        self::$dataStream=$newDataStream;
        return true;
    }
    return false;
}
//______________________________
public function getRandomDataRow(){
    $aData=new DataRow();
    $aData->setMyContent($this->randData);
    return $aData;
}
//______________________________
private function randData(){
    return 1.421;
}
}//end class
```

Figure 4.13 The `DataReader` class in our sample data reader system.
// instantiate a new data-stream object
$stream = new DataStream();
$stream->setMyResource("/mysql:3306");

// instantiate a reader;
$reader = new DataReader();
$reader->authenticated = true;

// set stream to the reader,
$success = $reader->setDataStream($stream);
if ($success)
{
    $dataObj = $reader->getRandomDataRow();
    echo "A Data Object: ", $dataObj->getMyContent();
}

Figure 4.14 Statements in the main script to show a sample instance of the data reader.

1. objChangedState("Object instance: var(stream) changed state in a func. or method body @ lab(59).")
2. typeCoercionWarning("Possible Illegal Type Coercion @ lab(84). Got: Float(). Expected from: String()")

Figure 4.15 Warnings generated when running PType on script in Fig. 4.14.

```php
public function setMyResource($val)
{
    // if (!($self::resourceSet))
    if (!($this->resourceSet))
    {
        $this->myResource = $val;
        $this::resourceSet = true;
    }
    return $this;
}
```

Figure 4.16 Changing the access identifier for a static class property.

```php
"DataStream.setMyResource"
1. accessUndefinedClassProperty("Attempt to access un-defined object property: var(resourceSet) @ lab(12).")
2. usingVoidTypeForExpr("Computed typeSet for lab(12) has Void type.")
3. typeCoercionWarning("Possible Illegal Type Coercion @ lab(12). Got: Void(). Expected from: Bool()")
```

Figure 4.17 Warnings generated in response to error in Fig. 4.16
4.3. ERRORS IN CLASS EXPRESSIONS

CHAPTER 4. EVALUATIONS OF ERRORS

```php
// instantiate a new data-stream object
$stream = new DataStream();
$stream->setMyResource("/mysql:3306");

$reader = new DataReader();
// user forgets to authenticate the reader
// $reader->authenticated = true;

$success = $reader->setDataStream($stream);
```

**Figure 4.18** Accessing an uninitialized property

```php
<"DataReader.setDataStream"> 1. readUninitializedClassProperty("Attempt to read un-initialized object property:
                                              var(authenticated) @ lab(29).")
  2. usingVoidTypeForExpr("Computed typeSet for lab(29) has Void type.")
  3. assignVoidTypeToVar("Assigning a Void type to var(cond) @ lab(26).")

**Figure 4.19** Changing the access identifier for a static class property.

```php
....
.
$success = $reader->setDataStream($stream);
if ($success) {
    $dataObj = $reader->getRandomDataRow();
} else {
    $dataObj = "Bad Object";
}
echo "A Data Object: ". $dataObj->getMyContent();
```

**Figure 4.20** Invoking a method call on a non-object variable.

```php
1. callMethodFromNonObjectVar("Attempt to call method: getMyContent from non-object @
                                  lab(91).")
  2. usingVoidTypeForExpr("Computed typeSet for lab(91) has Void type.")
  3. varHasMultipleTypes("Var dataObj acquired more than one types at lab(82).")

**Figure 4.21** Warnings produced in response to calling a method on a non-object variable.

101
4.3. ERRORS IN CLASS EXPRESSIONS

CHAPTER 4. EVALUATIONS OF ERRORS

```php
class DataReader{
    private $authenticated;

    public function setDataStream(){
        $cond=$this->authenticated;
    }
}

//mistakenly drop 'd' from property name.
$reader=new DataReader();
$reader->authenticate=true;
```

**Figure 4.22** Dynamically adding a new class field.

```php
"ClassRelatedErrorsIntroduced4.php"
  1. dynamicFieldInsertion("Adding field: var(authenticate) to object @ lab(70).")
  2. ...
"DataReader.setDataStream"
  1. readUninitializedClassProperty("Attempt to read un-initialized object property: var(authenticated) @ lab(31).")
  2. ...
```

**Figure 4.23** Prominent warnings generated in response to the change in Fig. 4.22
4.3.5 Using the $this identifier in a static context.

Static methods are not associated with any one specific object spawned from the parent class. As a result, any reference to the $this identifier in a static method body is a programming violation and will cause a fatal runtime error. We’ve also introduced this type-checking feature in PType. An example is shown in Fig. 4.24.

```php
class DataReader {

    public static function validateStreamAndAuthentication($dataStream) {
        if (($this->authenticated) && ($dataStream!=null)) {
            return true;
        } else {
            return false;
        }
    }

```

Figure 4.24 Using $this in a static method body.

In Fig. 4.24 we’ve defined a new static function: `validateStreamAndAuthentication` that seeks to perform some form of validation. The function body includes a reference to the $this identifier, which causes the PHP compiler to end execution because it is unable to resolve the object bound to the $this parameter. PType preemptively detects such a scenario, and reports a warning to the user. The warning is displayed in Fig. 4.25.

```xml
"DataReader.validateStreamAndAuthentication">
  1. referenceObjInStaticContext("Attempt to use this in static context @ lab(62).")
  2. usingVoidTypeForExpr("Computed typeSet for lab(62) has Void type.")
  3. readPropertyFromNonObjectVar("Attempt to read property: authenticated from non-object @ lab(63).")
  4. ...

```

Figure 4.25 Warnings generated in response to calling the static method defined in Fig. 4.24.

It is interesting to note an additional kind of type-checking feature from the warnings generated in Fig. 4.25. The type-checker internally binds the $this identifier with a non-object type (Void) in the
4.4. CONCLUSION

The expression:

\[ \texttt{this} \rightarrow \texttt{authenticated} \]

is, in the meanwhile, an attempt to access a property from the object associated with \texttt{this}. PType detects the error in trying to retrieve an object property from a non-object type, and reports the \texttt{readPropertyFromNonObjectVar} warning (Warning #3, Fig. 4.25).

4.3.6 Additional warnings with class-related expressions.

PType offers a few additional type-checking features aside from the ones already mentioned in this section. For example, when an object instantiation expression (i.e. \texttt{new ClassName()}) is encountered for a class whose definition is not available, it (PType) issues a \texttt{classDefNotFound} warning. The warning also includes information on the call site and the name of the object variable. Likewise, calling a method whose definition is not available, or calling it with fewer arguments than the pre-defined number of parameters will result in issuance of warnings as discussed in section 4.2. In all of these cases, and even more generally, in any scenario where a concrete type cannot be inferred because of a potential error in the script, PType issues a \texttt{void} type to the erroneous expression and continues with the analysis.

4.4 Conclusion

In this section, we reviewed the variety of type checking features that are supported in our type checker, PType. We acknowledge that this set of type checking features is not exhaustive, and that there is room for enhancements. However, PType already carries a comprehensive state data of the script under scrutiny. This data can be subjected to a wide variety of analyses to come up with any kind of needed type-checking features. The given set of type checking features is a small sample of this capability. We hope that this set of basic type-checking features can be used as a foundation for building more complex analyses.
The work in this thesis can be recommended for many future improvements. This work can be used as a platform to build additional analysis components. The ultimate goal may be to provide type and error analysis support for the full PHP language specifications. While this may seem quite ambitious, we provide discussions on some short term future goals.

The analysis currently can't type-check more advanced object-oriented features. This involves, but is not limited to, features involving inheritance, interfaces, traits, and other object-oriented components supported by PHP. Including them in the analysis could be done with a relatively small amount of effort. For instance, the introductions of inheritance, interfaces, and traits should not require modifying our object type in any way. Their usages simply augment the number and kind of members (methods and properties) that an object instance may have access to. Additional, and some arcane, PHP features such as magic methods, overloading, and object iteration may required some more involved integration. Regardless, we believe that the provided platform is quite adequate to support analysis for all these PHP features.

Another way to augment the analysis would be to track specific data stored in some variable, or
CHAPTER 5. SUMMARY AND FUTURE WORK

generated by some expression of interest. For instance, we discussed in chapter 3.3.9 that it is possible to include files dynamically. In such a case, it doesn't suffice to only have the types for the variable/expression used. The explicit value stored or generated in that variable/expression is also required to figure out the exact source file to include. The same is true for situations such as the following:

```php
<?php
    $a = "b";
    $$a = "Hi!"; // instantiates a new var "b"
    echo $b; // prints "Hi!"
?>
```

In the above, the specific data in $a is used to instantiate the program variable $b. Thus, it doesn't suffice to say that $a simply stores a str type in the type analysis. We need to also be able to retrieve the string "b" from $a, so that later in the analysis, we can create var $b and use it.

A useful addition that could be built is a top-level layer to process the warnings generated during a type analysis job. Currently, the type analyzer is designed to gather all kinds of warning without discrimination. This could often result in a long list of warnings, especially due to error propagation. This top-layer processor could, with some help from the underlying framework, discern against such cascading warnings.

Also, each warning could be assigned a severity index, and only qualifying indexes over a threshold could be displayed. Support for additional user-defined settings could be put on place to turn off/on several kinds of warnings generated during the analysis. Because the type-checker is still under development at the moment of this writing, it is to our benefit to emit verbose warning sets for debugging purposes.

Finally, Rascal has a good integration with the eclipse editor. This means that the generated warnings could be used to directly annotate source code if used with an eclipse compatible PHP tool. This may provide a much more usable interface for the type analyzer than it currently features. For that purpose, we're currently exploring the possibility of integration with the PDT (PHP development
tool) plugin for eclipse.

The work on this thesis provides an implementation of a proof-of-concept tool that we've demonstrated can be used for many kinds of error analysis in PHP. The tool is based on a rich constraints-based language, and provides precise type-inference for many dynamic features of PHP. The core of this method itself is language-agnostic, and could hypothetically be applied to any dynamic language. We've open-sourced PType [Sof], including the PHP-Air framework that it depends on, and encourage enthusiastic readers to explore this tool in terms of their own interests.


[Mil78] Milner, R. “A theory of type polymorphism in programming”. *Journal of Computer and
BIBLIOGRAPHY


[Phpb] [www.php.net](http://www.php.net).
In the following pages, we list out an example PHP script, the control flow graph generated by PHPAir for this script, the systems of generated constraints, and finally a typed flow model for this script using PType. Readers are advised to refer to these contents while perusing through the technical aspects of this work.
<?php

function some_func($x, $y) {
    ($z = array($x, $y))
    (return $z)
}

($a = 7);
($b = 0);

if (($a % 3) == 0) {
    ($b ++);
} else {
    ($b = "foobar");

($c = some_func($a, $b))

?>

Figure A.1 An example labelled script.
APPENDIX A. ILLUSTRATIONS OF ANALYSIS COMPONENTS

Figure A.2 System of constraints generated for the script in Fig A.1.
Figure A.3 CFG of the script given in Fig A.1.
Figure A.4 Typed flow model of the script given in Fig A.1 (Continued...).
Figure A.5 (Continued Fig. A.4) Typed flow model of the script given in Fig A.1.