Thesis for the degree of Master of Science

Soft typing and analyses on PHP programs

Patrick Camphuijsen

Supervisor: prof.dr. S.D. Swierstra
Daily supervisor: dr. J. Hage
Utrecht, March 2007
INF/SCR-06-37

Department of Information and Computing Sciences
Universiteit Utrecht
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Universiteit Utrecht

Abstract

PHP is the most popular scripting language used for the design of websites on the Internet. Such languages often are quite vulnerable, due to the fact that no internal checks for typical mistakes such as misspelled variable names are made. Another problem is the fact that several functions perform the same task, yet have very different names, and documentation is inconsistent. Also, variables can be reused several times in a PHP script, often unintended, resulting in undesired behaviour of code.

Several projects exist that try to detect, fix and report security vulnerabilities and incorrectness of PHP code. Most of these projects focus only on the security aspect. Therefore, validation tools are rather scarce and hard to find. Validation tools focus mainly on the HTML validation aspect, and therefore check on the output of a program.

Though this is useful, it does not provide any useful feedback for correcting these mistakes, as this validation has no link at all with the original code. Other projects check for certain bug patterns, but these patterns are merely focused on misconceptions, such as function calls with too many parameters, rather than e.g. specifically checking for undeclared variables. Problems such as these can only be solved properly with a type inference system, and this is not implemented by any of these tools.

In this master's thesis, we discuss the theories behind these problems, and discuss an analysis tool for PHP, that deals with many of these problems. We propose a type inference system that deals with aspects such as misspelled variables and type changes of variables, undefined functions, and (negative) effects that global and static variables could have on a program.

We also describe several other analyses, like dealing with HTML validation directly on the source code of a PHP script, correct handling of protocols such as database access, and finally a collection of coding styles, that could help to reduce the number of problems in a PHP script.

Contents

Chapter 1

Introduction

Websites are typically programmed in a *scripting language*, which means that their code is interpreted rather than compiled. These languages often have a gradual learning curve, which makes learning such a language easy. However, a major disadvantage of such languages is that mistakes made by a programmer are often hard to locate. At this moment, many scripting languages exist, of which PHP [44] is by far the most popular: statistics of November 2006 show that about 20 million and 1.3 million IP addresses use PHP [25].

PHP (acronym for PHP: Hypertext Processor) comes with a massive library of functions, constants and classes, that support string-, file-, image and database manipulation, combined with native shell access and text output capabilities. Most of PHP's functionality can be written in several different ways[1]. This, together with the *dynamically typed* nature of PHP, make it very easy to develop powerful applications in a very flexible language.

However, this flexibility is also one of the biggest disadvantages of PHP. Even though the library offers a lot of features and useful resources, it lacks consistency, and often the exact same functionality is offered by several functions. The dynamically typed nature makes it hard to find misspelled variables, because dynamic type systems often allow the use of undeclared variables. Furthermore, these problems combined with those that exist with HTML validation, make writing correct websites in such languages very difficult.

Also, many PHP sites have been written by inexperienced programmers, and are in a barely manageable state due to bad code design. Rewriting such a site can be tedious, and this is where *coding practices* come into play. By specifying simple, yet specific rules such as *'a function must print all of the closing tags for all of its opening tags'*, or *'a code block may not be larger than 30 statements'*, many of such problems can be identified, or even be corrected, without the need of manually reading through the entire code.

However, correctness of code is not the only problem that exists for websites. Many scripts access external resources, such as (externally located) files, socket connections, database connections and web form variables, in which the user of a website can enter information. Especially this last category is an apparent source of vulnerability, because if these variables are not validated, malicious users can use these fields to take control of a site to gain unauthorized access, by entering certain sequences of text, that allow them for example to take control of a database (SQL injections), or even control of the output of the site for a regular user (cross-site scripting, or XSS). More examples of security issues exist, but these are the two most important ones [33]. By identifying locations in a script where such *tainted information* can enter the program, and where this tainted information leaves the script (e.g. when written to a file or displayed on the browser), and by properly 'untainting' them, these security risks can be completely diminished.

These problems are not specific for PHP, and many researchers already addressed many such problems in PHP and similar languages. Static analysis is the most widely used solution to find suspicious constructs and coding patterns in source code. Many tools exist that can perform correctness and security checks on such languages.

[1] This means, some functionality is covered by several functions that implement the same thing, and some functionality offered by a set of functions, can also be implemented using object-oriented features, depending on the programmer's preference.

1.1 Problem description

To illustrate our problem, consider the following situation. Imagine a script that takes a message from a user, then prints it, stores it in a database, and finally shows all of the messages stored in this database.

```
<form action="<?php echo $PHP_SELF; ?>">
<input type="text" name="message">
<input type="submit" value="go">
</form>
<?
if ($HTTP_GET_VARS['message']) print("Message: " . $HTTP_GET_VARS['message'] . "<BR>");
mysql_connect("mydatabase.com","myuser","mypassword");
mysql_select_db("mydatabase");
mysql_query("INSERT INTO Messages (MESSAGE) VALUES ('$HTTP_GET_VARS['message']')");

echo "<H4>Old messages</H4>";
$result = mysql_query("SELECT MESSAGE FROM Messages");
$i = 0;
while ($row = mysql_fetch_row($result)) {
  if ($i%2 == 0) echo "<UL>"; else echo "<OL>";
  print($row[0]);
  if ($row[0] == "0") $i++;
  if ($i%2 == 0) echo "</UL>"; else echo "</OL>";
  $i++;
}
?>
```

Several problems exist in this script, which we will briefly explain and discuss. To start with, this script uses a *global variable* named `$HTTP_GET_VARS`, which gives access to web input variables that have been passed along the URL of a website. However, this particular variable has already become deprecated in PHP, as it has been replaced by the `$_GET` variable. Furthermore, these web input variables are always strings, yet it has been used here as a boolean condition in the `if` statement, which is possible because PHP automatically casts values to the type that is needed in a given situation.

These *coercions* often occur without problems, but there are situations in which this can definitely result in incorrect or unintended behaviour[2]. In the code example it does not cause any problem, because the code relies on the behaviour of PHP that non-empty and undefined strings are converted to `false`, and other strings to `true`. The same can be said about the `while` loop, but here the `mysql_fetch_row` actually returns `false` if there are no more rows in the query result set.

Another problem in this code is that there is a security leak; the contents of the `$HTTP_GET_VARS['message']` web variable are not validated anywhere, but directly printed to the screen. A malicious user could use this vulnerability to his own advantage. Also, this variable is used in a database query on the next line which, without any validation, would give a malicious user unrestricted access to any part of the database. If he enters a message such as "`'; DELETE * FROM Messages`", all of the messages in the `Messages` table would be deleted.

Before the `while` loop, the text `<H4>Old messages</H4>` is printed, which shows the text "Old messages" in a small header font. However, these tags are in uppercase, while the current XHTML standard [10] demands that all tags are in lowercase. In the `while` loop itself, the printing of `<ul>` and `<ol>` tags are alternated by the use of a counter variable `$i`. However, between the `print` statement and the statement that prints the closing tag, the value of `$i` could change, which could result in printing the wrong closing tag. And again, these tags should be in lowercase.

Another potential problem is the inconsistent use of the `echo` statement and the `print` function. It looks very unclean and confusing if the same functionality is used through different statements and functions. Though this is only a simple example, it would get even more confusing if for example the functions `strpos($haystack, $needle)` and `in_array($needle, $haystack)` are mixed up, which both look up an element in an array (strings could be considered as arrays of characters) of elements.

[2] For example, consider a string to number coercion: not all strings represent numbers!

Another problem lies in the database handling. For none of the queries, a call to the `mysql_free_result` is performed, which frees up the memory used by that query. Also, a call to the `mysql_close` function is missing, which closes the database connection.

1.2 Goals

The problems described in Section 1.1 are only a simple example of things that can go wrong in a PHP script. In this master's thesis we explore the problem domain of validation of PHP code, give examples of evident situations for which the need of validation exists, and offer solutions that perform this validation, in the form of static analysis on PHP code.

We recognize three different sub-domains: HTML validation, coding practices and protocol handling, and type inference. Each of these sub-domains requires a different approach.

HTML validation, for example, is typically performed on the output of a website. However, such a validation does not give any information about where the specific problems exist in the source code. Using a (regular) HTML validator therefore requires a programmer to skim through his code manually, to try to find and solve the problems listed by the validator. We explore the possibilities of performing such a validation directly on the script's source code, so that not only the same problems can be inspected that a HTML validator does, but now we can also provide better feedback, based on the locations in the source where the problem really originates from.

Many websites use a database, and follow a typical protocol for making a connection, performing queries, and closing the connection. Forgetting a (crucial) step in such a protocol may affect the entire program, and finding the location where such a mistake is made, can be a lot of work. We have defined analyses that check for the correct use of such a database protocol, that report where illegal operations are performed. Other protocols, like file access and sessions, follow a similar pattern.

Coding practices are mainly based on programming styles. We assume that not every programmer writes his code in the exact same style. Coding practices can be used to check the consistency and readability of a code fragment. For example, a coding practice could check if a function prints all of the HTML closing tags for the opening tags that are printed, or if a code block does not exceed a certain amount of statements. These coding practices can be formulated for example as simple criteria that a code fragment must meet, or as a set of actions that are allowed or disallowed, for any situation.

Because PHP is a dynamically typed language, a static type system will not work. A static type system would reject any ill-typed program, while the dynamically typed PHP interpreter does not reject any program at all (unless it throws a fatal error; but even then the program is *only* rejected after this error is thrown). However, using a more tolerant type system, i.e. *soft typing* [7, 2, 45], it is still possible to define a type inference system for PHP that respects the dynamic nature of the language, yet still imposes enough restrictions on the language to ensure that suspicious fragments can be identified and reported.

The results of this master's thesis project are this master's thesis that describes the theories behind these three sub-domains, a type inference system that specifically works on the PHP language, and a prototype tool that performs analyses for the sub-domains described above, including the type inference system, and reports the results that it has found back to the programmer as a list of warning messages. These warning messages are further categorized on subject, but also on priority (i.e. how serious a warning actually is), and can be suppressed, if the programmer is not interested in a specific classification of warning messages.

We do not strive to automatically change any PHP code with our tool; our focus is on warning a programmer that a certain piece of his code contains *possible* mistakes. The programmer could use this information to make his code more trustworthy, by making simple changes to his code, or even by rewriting it in such a way that the potential problem is avoided. Of course, not all of these warnings are actual errors. The number of these *false positives* however we try to keep as low as possible, e.g. by using suppression lists to hide them. The tool itself uses a framework that is not specific for PHP only; it can be extended and adapted to work with other (scripting) languages as well.

1.3 Organisation of this thesis

This thesis is organised as follows. We start with an introduction of several dynamically typed languages, to give an idea about what kind of language we are discussing (Chapter 2). Then, we discuss the known theory of each of the three sub-domains described in Section 1.2: type inference and soft typing (Chapter 3), HTML validation (Chapter 4), and coding styles and protocols (Chapter 5). We discuss the literature and theories, and existing solutions for each of the languages that we described in Chapter 2.

In the second part of this thesis, we focus on the PHP language. We start with a more elaborate description of the PHP language and its features (Chapter 6), followed by the soft typing system we developed (Chapter 7), and the other analyses that we define within our system (Chapter 8). Next, we describe our prototype tool PHP Validator that implements some of these analyses (Chapter 9), and explain our framework. Next, we test this tool on several sets of programs, and discuss the results (Chapter 10). Chapter 11 concludes.

Chapter 2

Dynamically typed languages

Websites are usually programmed in scripting languages, such as Perl [42], Python [36] and PHP [44], which return a page formatted in a text-based structured document, e.g. in HTML. Often JavaScript [19] is used as well, which is a client-side scripting language which can be used to enhance plain HTML, or to support the server-side scripting code. Users of websites programmed in such a language never notice the actual code of these sites, just the text output they deliver. Behind such a site a whole collection of programs that controls input and output is hidden, written in one of these languages, complete with constructs like *if-then-else* branches, *while* loops, and other constructs found in everyday programming languages.

Scripting languages are often *dynamically typed*, which means that the programmer does not have to declare variables before using them, and the types of these variables can change throughout the program. And, the values of these variables can e.g. be subjected to a coercion (like a string suddenly being used as an integer, or vice versa). This can be dangerous unless guarantees are made that these coercions do not lead to erroneous code[1], or unpredictable situations at run-time. The meaning of erroneous code here is that it does not reflect the actual intentions the programmer had when he wrote it; for example, some languages treat every variable as a string, and only when needed convert it to an integer. If this is not possible, no error is thrown, but instead, the converted string is represented by the integer value of 0. Such situations are often unintended, and very hard to discover unless the code is thoroughly tested by the programmer.

In this chapter, we consider four scripting languages, discussing their main constructs and abilities. Because there are also dynamically typed functional languages, albeit strongly typed, like Scheme [16], we will consider this language as well.

2.1 Perl

Perl is a "Practical Extraction and Report Language" [42] available for a wide series of operating systems. Perl is designed mainly for text manipulation, and offers powerful constructs for regular expressions to process (large) documents of text quickly. Perl borrows its features mainly from C, but also shell scripting (sh), Lisp and many other programming languages.

Perl has three fundamental data types: scalars, lists, and hashes. Scalars are the primitive values programmers are familiar with (integers, strings, but also pointers in this case), lists are ordered collections of scalars (practically equivalent with arrays), and hashes, or associative arrays are key-value maps from strings to scalars. Perl has several control structures, like block-oriented control structures, e.g. `if ( cond ) { ... } else { ... }` for conditional block execution, as well as `while` loops in various forms. There is also a `foreach` construct, that allows a collection structure to be iterated over.

Perl has the notion of functions, called subroutines, that can be invoked anywhere in the program. Arguments of a subroutine do not need to be declared as to either number or type; in fact, they may vary from call to call. Arrays are expanded to their elements, hashes are expanded to a list of key-value pairs, and the whole list of arguments itself is passed to the subroutine as a list of scalars, called `@_`. Subroutines can return multiple

[1] Coercions are implicit yet can have quite an impact on a program. A coercion could fail and result in an unexpected result value, and this makes them dangerous.

values using the `return` statement, or this statement can just be skipped, in which case the value of the last statement of the subroutine is considered the return value.

The real power of Perl lies in the use of regular expressions [13]. A specialized syntax has been implemented, and the interpreter has an engine for matching strings to these regular expressions. This engine uses a backtracking algorithm allowing simple pattern matching to string capture and substitution. These regular expressions can be used to find text in a given input, count occurrences, validate it, or to even transform the text.

Perl also has good facilities for database applications. Its text handling facilities are good for generating SQL queries, while arrays and hashes are very useful for collecting and processing the returned data. Special Perl DBI (Database Interface) modules are available, and DBD (Database Driver) modules control the access with over 50 types of different databases. New database types just require the addition of a new DBD.

Here is an example of Perl that demonstrates the use of scalars, lists and hashes and subroutines:

```
sub list  {      (4, 5, 6)     }
sub array { @x = (4, 5, 6); @x }

$x = list;   # returns 6 - last element of list
$x = array;  # returns 3 - number of elements in list
@x = list;   # returns (4, 5, 6)
@x = array;  # returns (4, 5, 6)
```

2.2 Python

Python [36] is a language that combines multiple paradigms like object-oriented programming and functional programming into one language, and offers extensions to support even more. One important feature in Python is dynamic name resolution, which binds method and variable names during execution. Python is mainly developed with the aim to improve coding practices and highly readable syntax, in contrast to Perl, in which program code can look cluttered, and very unreadable for those not familiar with Perl (or even Perl programmers who wrote the code themselves). Python can both be used as a scripting language and for application programming.

In Python, whitespace actually does have syntactic meaning, similar to Haskell. Curly braces are not used, but instead indentation is used to denote different program blocks from control sequences. The main disadvantage here is that both tabs and spaces are considered as distinctly different indentation, but most editing programs do not make this distinction. In some cases even, tabs are converted to spaces (and not all editors use the same number of spaces for each tab), which may destroy the complete syntax of a Python program, if not carefully used.

In Python, data structures and values are organized in a way that variables only hold references to objects. There are base types, which can contain the usual primitive values such as numbers and strings, collection types for sequences and key-value mappings, and an object system supporting sub-classing and multiple inheritance through mixins.

Python also allows a functional programming style, making working with lists and other collection types more straightforward. This includes features such as list comprehensions and first-class functions known from languages such as Haskell, but also *closures* (functions created at run-time), and *generators*, which is Python's equivalent for lazy evaluation.

The follow example illustrates some of Python's string features:

```
#!/usr/bin/python

# Strings have various escapes.
print "Hi\nth\ere,\thow \141\x72\145\x20you?"

# Raw strings ignore them.
print r"Hi\nth\ere,\thow \141\x72\145\x20you?"
```

2.3 JavaScript

Unlike the name may suggest, JavaScript [19] does not have much to do with the Java programming language itself. This language is a prototype-based scripting language with a syntax loosely based on C – in fact both Java and JavaScript both came forth out of C independently –. Originally it was called Mocha, then LiveScript. The JavaScript name may come from the time that Netscape decided to add support for Java applet technology in their browser. There's also ECMAScript, of which the fourth edition forms the basis for the upcoming JavaScript 2 edition, which is going to be released in 2007.

JavaScript relies on a host environment, for example HTML pages, and can interact with the Document Object Model (DOM) of the page to perform tasks that cannot be done with plain HTML, like opening pop-up windows, or checking/validating input values to make sure they are acceptable *before* being sent to the server.

The syntax of JavaScript is somewhat different from Java's syntax. Since Java is statically typed, types have to be given to variables at declaration. In JavaScript however this is not necessary. You can leave out the declarations of variables to make them global variables (and you can create them inside functions, like in Perl), or local variables, by using the `var` keyword. Arrays work differently as well, you can refer to any index with either a string, or an integer value, but alternative representations of a number, like `"01"` for the position with index `1` are illegal. Another difference between JavaScript and Java's syntax is the fact that strings can be defined between both single and double quotes in JavaScript. Objects in JavaScript also work differently: they are structures with a set of key-value pairs, where values can be objects as well. Any type can be assigned to any object.

The biggest downside with JavaScript is security and debugging. Since JavaScript is interpreted on the page it has been defined on, errors can only be detected at run-time, and interaction with the DOM may also cause problems, since it may differ among browsers. Security problems are often of the 'confidential information stealing' kind, using techniques such as *cross-site scripting*, for which a client-side language such as JavaScript is more vulnerable than server-side languages.

One application for which JavaScript is often used, is client-side input validation. This is useful to help finding bad inputs before it reaches the server (though it does not release you of the need to do the validation on server-side also). Here's an example that performs such a validation:

```
<script language="JavaScript" type="text/javascript">
<!--
function checkform ( form ) {
  if (form.email.value == "") {
    alert( "Please enter your email address." );
    form.email.focus();
    return false ;
  } else
  return true ;
}
//-->
</script>
```

2.4 Scheme

Scheme [16] is a functional programming language, and a dialect of Lisp. Its design is to be as minimalistic as possible, which means that its structure is as simple as possible.

Since Scheme is a dynamically typed language, variables have no static type. Variables are bound by a *define*, a *let* expression, and a few other Scheme forms [16]. These constructs also control the scope of these variables. Variables can also contain functions, which are treated as first-class objects in Scheme. Functions are divided into two basic categories: procedures and primitives. Primitives are predefined, and contain simple operators like `+` and `-`. Procedures are the user-defined functions, and can use the primitives and other procedures (in fact, all primitives are procedures too, but the opposite is not the case).

Lists are the primary data structure in Scheme, and are organised as linked lists. Lists can contain values of any of the primitive data types (numbers, booleans and strings), and also other (heterogeneous) lists. Using

the functions `car` and `cdr`, the head (first) element of the list, or the tail (last elements), can be selected respectively.

There are three types of equality in Scheme: equality with regards to variables pointing to the same data object in memory, equality for numbers and objects sharing the same memory location and for complex data structures like strings, lists and vectors, and lastly there is type dependent equality for strings, characters and numbers.

Control structures in Scheme are the usual `if-then-else`, and a more general one called `cond` which allows conditionals in the way such as `case` and `switch` constructs in languages such as Java work, but here it works without fall-through [2], so it is more structured in Scheme. Loops take usually the form of tail recursion [16], or use the `do` construct that basically works like an iterator.

The following example shows how Scheme can calculate the sum of all values in a list:

```
define sum
  (lambda args
    (sum-of-list args)))

(define sum-of-list
  (lambda (lon)
    (if (null? lon)
0
(+ (car lon)
   (sum-of-list (cdr lon))))))
```

2.5 PHP

PHP [44] was originally designed as a set of Perl scripts, and a set of CGI binaries written in C, with the name "Personal Home Page Tools". Later this name was changed to "PHP: Hypertext Processor", and soon the language become popular among web programmers. PHP is often seen as an alternative to languages such as ASP.NET and VB.NET, as these are also server-side languages and work very similarly. There are many publicly available popular PHP applications to provide e.g. a forum and wiki system to a website. The PHP version considered here is PHP 5.0.

PHP stores numbers (both whole and floating point) in a platform-dependent range, which means that e.g. integers on one system running PHP can be 64-bit, whereas on another system they are only 32-bit. Portable code should therefore use numbers in the 32-bit range to ensure it works on all systems. PHP has the notion of booleans as well, which basically means that every numerical non-zero value is considered as *true* in a boolean expression, and zero as *false*, similar to in Perl and C. The empty string `""` also evaluates to *false*. There is also a Null data type, which represents variables that are without a value.

Arrays in PHP are heterogeneous which means a single array can have values of different type, including user-defined objects. Also associative arrays are allowed, which are a generalization of normal arrays (which are in fact associative arrays with integer values as the key values). When used as associative arrays, ordering of keys is preserved. Strings in PHP can be specified inside single quotation marks, as well as double quotation marks, though only double quotation marks evaluates variables specified inside them. This is called interpolation. The PHP interpreter internally translates these strings to concatenations of strings inside single quotation marks. An example:

```
<?
$a = "cd";
$b = "ab $a ef";
print($b); //"ab cd ef" is printed
$c = 'ab $a ef';
print($c); //"ab $a ef" is printed
?>
```

[2] This means that from one `case` construct in Java, you can drop into the next one, since these are not closed blocks of code, but the `switch` construct with its `case` statements is more some kind of jump table.

A resource type exists in PHP as well, which are basically references to resources from external sources (e.g. images and databases). Only functions from the libraries of these resources can process them. For example, a MySQL database link can only be processed by the MySQL extension library for PHP.

Objects in PHP are of the more simple kind, offering fields, methods (procedures returning no value), and functions (procedures that do return a value), which have the same meaning as functions defined on the global level of the program. Objects are referenced by handle, which are equivalent to a pointer to their specific place in computer memory (in older versions they were referenced by value). Class members may be private and protected, and there may be abstract classes and abstract methods as well. The declaration of constructors and destructors of a class is done similarly to other object-oriented languages, such as C++. Another feature that is available is exception handling, which also works quite the same way as in languages like C# and Java.

In Chapter 6 we will give a more detailed specification of the PHP language, its constructs, and its features.

Chapter 3

Type systems

Type systems are designed to prevent the improper use of program operations on variables and functions, e.g. multiplying a string with an integer. There are two main classifications of type systems: *static*, and *dynamic*. Static type checking occurs at compile-time, dynamic type checking occurs at run-time. Both will detect ill-typed programs, and report the found problems.

Static type systems have two important advantages over dynamic ones: they provide feedback to the programmer by detecting a large number of program errors before actual execution, and they can extract some information that a compiler can exploit to produce more efficient code. However, the disadvantage here is a loss of expressiveness, modularity, and semantic simplicity. Static type systems are used in situations where the types of variables are to be determined before run-time, whereas dynamic type systems discover the types during run-time. Often, a language using a dynamic type system also allows the types to change during execution, which is difficult to allow or impractical with a static type system. Dynamic type systems assign types to variables usually at the first program point where they occur and change them if necessary when the same variable is assigned a different value. Languages using static type systems require variables and their types to be declared first – or infer the type if omitted –, and then keep the same type for these variables throughout the entire program (fragment). So, changing the type of a variable in a single execution of the same program fragment is impossible in a static type system.[1]

3.1 Hindley-Milner type inference

Type inference systems are often modeled after Hindley-Milner's system [37], which uses a system of *type variables*. Each time a new variable, or function is encountered, a new unique type variable is assigned, and based on a system of logical deduction rules, the exact type of each variable or function can be calculated. Because there are primitive types for each kind of value [2], variables and functions can be traced to these fixed primitive types. Some functions though, e.g. *map* in most functional languages, can be used on multiple types, and thus are *polymorphic*. In Hindley-Milner's system every type containing one or more quantified type variables is called polymorphic.

Type inference systems modeled after Hindley-Milner consist of two important parts. Firstly, a set of deductive inference rules, that allows us to determine whether a given program and type assignment is type correct, and secondly an algorithm that takes an abstract syntax tree (AST) of the program, and calculates the actual or most general types using the information from the sub-trees to infer the type of the more complex fragments of the tree.

Algorithms for Hindley-Milner type inference may differ considerably, and only the original algorithm $\mathcal{W}$, as explained by Milner [37] are considered here. This algorithm achieves its goal by first generating a list of equations for each program fragment, and then runs a unification algorithm that applies substitutions in the equations on places where types are equal. A detailed explanation of this algorithm can be found in [37].

[1] Polymorphic type systems can be considered as an exception to this. They allow execution of the same program fragment with different types for variables, for as long as it remains sound. For example: `let id = \ x -> x in id id` where `id` has different types, although they are all instances of the same type scheme.

[2] This does not count for every language. Haskell for example has overloaded numerals. This means that until an exact type for a numeral value is filled in, the actual type will not yet be known. Until that moment, an overloaded type $Num\ a.a$ is used.

There are also languages that use a static type system other than Hindley-Milner's. An example is Java, which requires a declaration of all fields and local variables, and then just checks everywhere throughout the program if the types of these variables match their uses in expressions. There is no type inference for declarations.

Another element that is not present in Hindley-Milner type inference, is the notion of sub-typing (e.g. subtypes of objects and classes in an object-oriented language, or integers as a subtype of floats). There are extensions to the Hindley-Milner type inference systems that handle this, but are not part of the original system.

3.2 Soft type systems

Even though they have many advantages and can improve the quality and confidence in programs, static type systems impose a lot of restrictions on the programming language, causing loss of expressiveness and flexibility. To ensure type safety, programs that do not meet the strict requirements of the type system are rejected for execution. In rejecting such untypable programs, the type system may also reject meaningful programs that it cannot prove to be safe, and programs that are equivalent to these are often much more complicated. *Soft type systems* [7, 2, 45] provide the benefits that static type systems have to offer, combined with the flexibility that dynamically typed languages have. Like a static type system, the soft type system infers types for variables and expressions, but instead of rejecting programs with untypable fragments, it inserts explicit checks in the code that transform these fragments to typable form, and/or reports a list of situations with possible problems, which the programmer can then use to improve the quality of his program. Soft type systems can minimize the number of run-time checks in the code, allowing dynamically typed languages to gain the efficiency of statically typed languages. In short: soft typing is a generalization of static and dynamic typing.

The key technical issue for making a soft type system is to ensure that:

- It is rich enough to ensure that most programs that are written in dynamic typing style are well-typed;
- It is simple enough to accommodate type inference like in static type systems.

In [7], two main criteria are set for a soft type system to be effective:

Minimal-Text-Principle The system should not require any type declarations of the programmer for any program operations.

Minimal-Failure-Principle The system should leave as much program components untouched, unless they can cause type errors during execution.

The first condition can be met by accommodating *parametric polymorphism*, a form of modularity found in all dynamically typed languages. An example of parametric polymorphism can be found when calculating the type of a common function such as *map*, found in most functional languages. *map* takes a function of type $\alpha \rightarrow \beta$, and a list containing values of type α. For any α, β, *map* maps the type list(α) to list(β). To propagate the correct type information of a specific application of this *map* function, we need to capture the fact that the elements of the output list are of the same type as the result output of the function that is passed to *map*, otherwise we can't type check subsequent operations on the elements of this output list. So, a soft type system must be able to express that *map* has the type $\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow \text{list}(\alpha) \rightarrow \text{list}(\beta)$.

However, a type system based only on parametric polymorphism will not satisfy the minimal failure condition as well. There are two classes of program expressions that often occur in dynamically typed programs that do not type check with just parametric polymorphism. The first class is the set of expressions that do not return uniform results, i.e. for example `if-then-else` expressions that return a value that can be of different types (e.g. one branch returns a string, the other returns an integer). To assign types to these expressions, *union types* can be introduced. These types provide a union of two distinct types for each of these situations.

Another troublesome class of expressions are those that introduce recursive type constraints, like the self application function $\lambda x.(x\ x)$. Since x is applied as a function in the body of this example, it must have type $\alpha \rightarrow \beta$. But this self application enforces that $\alpha \rightarrow \beta$ must be the subtype of the input type of x which is α. Applying this to the identity function or to a constant would produce a well-formed answer, but most static type systems reject this. Solving this problem is possible by including a fixed-point operator in the language of type terms, which is called *recursive typing*. Another possible problem class that may be considered here, are heterogeneous lists in a language such as Scheme. Such lists can have values of different types, which may be lists themselves, and deciding on a type for these may often prove difficult. A combination of union types

and recursive typing could be applied here: union types for the different element types that may occur in the list, and recursive types for the elements that are lists themselves. It is even possible to just suffice with a list with a union type for the list elements, though this may result in reduced flexibility for deciding the exact type.

The type language for a soft type system must be able to handle the recursive typing and union types, and subsets thereof. For type inference, the Hindley-Milner system [37] would suffice, with extra rules for the sub-typing of union types, and for generic instances defined as in the paper of Damas and Milner [11]. A soft type system can also accommodate *conditional types* [2], with which the type of an expression e can be constrained using information about run-time tests in the context surrounding e. For example, in an expression `if` e_1 `then` e_2 `else` e_3, conditional types can express that e_2 is only evaluated if e_1 is true, and e_3 if e_1 is false. This requires that control-flow analysis on the program is performed to inspect these situations. Using control-flow analysis, it is possible to gain more accurate type information, as shown in [2].

What remains, is the insertion of run-time checks where required. Because a type checker cannot pass all good programs, it can be improved by adding run-time checks in places where it might go wrong, to ensure type correctness. A static type system would always reject these programs. However, a soft type system *may not* reject programs. It inserts run-time checks instead, at places where it detects otherwise untypable variables and functions, and make sure all errors are 'trapped'. To implement these checks, special functions called *narrowers* [7] are built. These narrowers perform the type checking, and return a value indicating whether or not the type check succeeded. This however will still not ensure that the programs will execute correctly. But, it will help the type system to adjust to the new situations in which the variables and functions are used.

As mentioned before, insertion of run-time checks can also be replaced by – as well as extended with – generating a list of warnings, for places in the code where possible problems may occur, such as variables whose type actually change at a given program point, or variables that may have either of a given set of types at some point. Such situations may not always result in an error, but they *are* possible weak spots, and deserve mentioning.

3.3 Other type systems

In the previous two sections we discussed type systems that make use of a type inference rule system, which can be used to infer the types for each variable (or function). However, the use of *constraints* [1, 41] for type inference is also a viable approach to create a working type inference system. Also, Hindley-Milner and soft typing usually make use of the control flow from the language. But in some cases, this may not be desirable. In this section, I'll discuss two systems: *iterative type analysis*, and *aggressive type inference*.

Iterative type analysis [8] infers the type of the body of methods, in three steps, by building a constraint graph. First, the variables are allocated, and form the nodes of the graph, and are of monomorphic type (polymorphism of variables is found through the iterations during the analysis). Next, the variables are seeded with their initial types, or a base type if no initial type can be determined. Lastly, constraints are created by drawing edges between the nodes and the operations on them. Iterative type analysis uses control-flow analysis to determine the types of variables, so that accurate types are determined at every specic point of execution in the program. Another type system described in [8], called Cartesian Product, performs type inference in a similar way, but on method calls instead. The algorithm is able to handle all possible flow constructs, and is flow-sensitive. This means that the inferred type of a variable may change as control flow is followed to a more restrictive type. Some situations however may require flow-insensitivity, which is supported as well. A full description of the algorithm can be found in Section 5 of [6].

Aggressive type inference [3] makes use of the idea that *"giving people a dynamically-typed language does not mean that they write dynamically-typed programs"*. Aggressive type inference ignores control flow; this means that at a specific flow construct, e.g. a *if-then-else* statement, the union of the result types of both blocks will be taken. The other rule is type consistency within a scope, which means that given a variable x with a type T at some point within the scope, it will have this type (or union) within the entire scope. Unfortunately, these two rules are not sufficient to infer types for some programs. Aggressive type inference however can be used in conjunction with other sources of type information, such as a list of predetermined types for some functions, so it can become more effective. Due to the fact that it does not recognize different types for a given variable at different program points, aggressive type inference is also not very accurate.

3.4 Applications in dynamically typed languages

In Chapter 2, we described five dynamically typed languages, and some of their features. Here, we will discuss what has been done in these languages on the area of type inference, and what kind of methods are used.

For Perl a static type inference system [31] has been developed. Because no formal grammar for Perl exists, aside from its implementation, the type checker uses the compiler back-end of Perl, to type check the variables and operators from the opcode tree which is generated, so no actual Perl source code itself is checked. The type system used here is a static type inference system which uses a unification algorithm similar to [11]. A soft typing system with union types like described in [2, 7] might have been preferable here, but instead, a solution [31] is chosen where type variables can be entered into specified places, so that things like unqualified numbers, generic scalars, and references to anything can be expressed without the need for these expensive union types. This however results in a type language which looks a lot more complicated than usual, but the added flexibility is expressive enough to yield the same results as union types would give, and is also more efficient.

This type language combined with the standard unification algorithm makes the type rules pretty straightforward. There are some difficulties in this system though; references treat type globs and references to type globs the same, and some of the Perl idioms, such as `do "filename"` cause the Perl compiler to compile code when the program is actually running, and so circumvent this type checker. Therefore, the type system for Perl described in [31] is unfortunately not completely sound.

For Scheme a soft typing system called Soft Scheme [46] has been developed. This system is modeled after the work of Cartwright and Fagan [7], which is basically the Hindley-Milner system extended with union types, recursive types, and sub-typing as a subset on union types. This soft typing system for Scheme, uses a more efficient representation for types, to integrate polymorphism smoothly with union types, and its run-time check insertion algorithm is more efficient, and inserts fewer checks. Also, some issues that Cartwright and Fagan ignored, such as uncurried procedures and assignments, have been addressed in Soft Scheme, making it a more practical and useful system.

Soft Scheme performs global type checking for R4RS Scheme [17] programs, and prints a list of the inserted run-time checks, after which the programmer can then inspect type information. For example, consider the following program, which is a function that flattens a tree to a proper list:

```
(define flatten (lambda (l)
                  (cond [(null? l) '()]
                        [(pair? l) (append (flatten (car l))
                                           (flatten (cdr l)))]
                        [else (list l)])))
(define a '(1 (2) 3))
(define b (flatten a))
```

When soft type checking this program, the type checker will give the summary `"TOTAL CHECKS 0"`, which means no run-time checks need to be inserted, since it is fully typable. Next, it will give the inferred types of the top-level definitions. When we would add `(define c (map add1 (flatten '(this (that)))))` to the program however, it will find an error, which will be flagged with the `ERROR-add1` function, which means that the program will always fail if this point is reached. Similarly, run-time check functions starting with `CHECK` will be inserted at places where checks are required, which is basically at most function applications.

For Python, several type inference systems have been developed, including the ones described in [3, 6], which are discussed here.

The system described by [6] uses a rather different approach than soft typing, namely *iterative type analysis*. The other type inference system is called *aggressive type inference*. A description of both can be found in Section 3.3.

For JavaScript a typing system is in development [18], for ECMAScript Edition 4, which was released late 2006, eventually leading to the JavaScript 2 release in 2007. Since this project is not finished yet at the time of writing this thesis, a lot can still change, but the idea of what they intend to create is quite clear already.

The proposal does not specifically describe an actual type checking system that does type inference externally, but more an internal (static) type system which is directly integrated in the language itself.

There are three primitive types, `#VOID`, `#NULL` and `#OBJECT`, union types that are unions of these primitive types, which are flattened when included in another union, and the elements of these union types that are subtypes of the types inside of union (a type is also a subtype of itself). Objects (and some of the other built-in types) can be made nullable (which means they can have the null value) or not; basically this means that their type is a union of the object's type with `#NULL` type. Logically this does not hold for numbers and booleans, since these are primitive types. There are other built-in types besides the three primitive types, e.g. `#STRING` and `#INT`, which are all direct subtypes of the `#OBJECT` type. There are class types, which describe the types for classes and interfaces, function types that describe the types of both unnamed and named functions, and structural object types, which are basically record-like types known from languages such as C.

Class, interface and function types can be parameterized, and such parameterized types can be seen as functions from types to types. The rest of the proposal is unfortunately sketchy and incomplete; more information can be found on [18, 22], which is frequently updated.

For PHP no type system has been developed so far, or could be found. This master's thesis will address this issue.

Chapter 4

HTML Validation

Scripting languages typically output a document in HTML format, that can then be checked by a HTML validator. However, a validator cannot give any information about where in the script's source code a semantic error was made, and how the HTML document was constructed. For example, a missing closing tag can be detected by a HTML validator, but in the source code itself it may look as if no mistakes have been made. Also, a missing closing tag in the HTML output can often be safely neglected, especially if it has no *visible* effect on the web browser's output.

The same can be said about using the wrong closing tag, though mistakes such as these are typically more visible. Though these may not have serious consequences, sometimes they can be disastrous. For example, a missing `</script>` tag may cause none of the JavaScripts on a website to run, and in some cases they can also result in a deformed page layout.

In this chapter we first describe some of the most common HTML validation issues. Next, we discuss some existing solutions for HTML validation that can be used in the languages that we discussed in Chapter 2. Because Scheme is not a scripting language used for building websites, it is not considered in this discussion. Finally, we discuss the advantages and disadvantages of validation of HTML by analysing the script's *source code* instead of validating the generated HTML output.

4.1 HTML validation issues

All HTML validators typically perform the same tasks: validation of a document's structure, its required and allowed tags, and its tag attributes. These validation tasks can be ordered in four main categories:

- Most tags (e.g. the `<table>` tag), have a corresponding closing tag. Sometimes these closing tags are not required for the well-formedness of the page layout (e.g. the `<p>` tag), but in other cases missing a closing tag may have serious consequences, e.g. missing a `</ul>` tag can deform the entire page. Common HTML validators therefore demand that *all* required closing tags are in the document, so that structural well-formedness is achieved as well. The tags that do not have a closing tag, must end with `/>` so that it is clearly visible that they do not have to be closed (i.e. they close themselves).
- Some tags, like `<blink>`, do not exist in the W3C specification of XHTML [10]. This, and other tags however only work in certain *specific* browsers. Finding such non-standard tags however is straightforward and trivial, by simply providing a list of valid tags - or a DTD to match the document with - to the HTML validator. Another task a HTML validator may perform is checking for typical syntax errors in the tags (e.g. uppercase characters). The same can also be done for tag attributes.
- Correctly nesting of tags is a third issue. A document that has the tags `<b><i>`, will expect the combination `</i></b>`, and not `</b></i>`. The last opened tag always has to be closed first, and then the second-last, and so on. To find such problems, a HTML validation can simply be done by a generic XML validator, that uses a DTD with all of the HTML specifics.
- Some tags have compulsory components, i.e. required inner tags or attributes that have to be present in each occurrence of the tag. Sometimes it will not have a visible effect on the page layout of the browser

if such a required component is missing. However, a HTML validator still demands such compulsory components to be present.

- There are two structural levels in a HTML document: *block-level*, which concerns whole blocks of text, and *inline-level*, which is a sub-level of the block-level, and concerns parts of a paragraph. This distinction of structural levels also implies that block-level tags *cannot* be nested. For example, the block-level tag `<p>` may not contain a block-level tag `<ul>`.

4.2 Existing applications in scripting languages

In this section we describe several solutions with regards to HTML validation in the languages described in Chapter 2. The examples described here are related to each of these languages, and show how HTML validation can be done directly by the script itself on its own output. None of these examples however provide a solution that helps to find invalid HTML by looking at the script's code itself.

In Perl, a library called `Test::HTML::Content` has been developed to deal with HTML validation [35]. Using XPath expressions for more complex searches, especially concerning HTML structure, it is possible to find most, or even all mistakes in the HTML code. One major downside of this library is that it has to reparse the entire HTML string whenever you call a function to test the HTML, which makes the testing of large documents rather slow. A possible solution to solve this, is by providing a caching mechanism that stores the most recent HTML. Another downside is that it does not actually report where things go wrong in the Perl code itself, only where in the HTML document the errors were found. However, applying such tests before actually sending the page output to the browser, or even applying the tests during construction of the page by building it up as a string, will make full validation of HTML code possible, before it is even printed. Still, using this library on code fragments (e.g. functions) that produce HTML, validation can still be more accurate than by performing on the entire output only.

The Webware system for Python [14] is a powerful system that not only provides HTML validation for Python scripts, but also an actual framework for working with servlets in Python, providing a significantly more powerful alternative to CGI scripts. Here, we only discuss its HTML validation properties.

The WebKit component [15] of Webware contains the actual classes and methods for dealing with generating dynamic web pages, using a servlet-like system similar to Java Servlets [30]. It includes methods that allow fast validation of the generated HTML output, before it is sent to the browser [15, 27]. The advantage here is that the HTML validation works automatically when it has been enabled; the disadvantage is that the source code of the script itself is not checked; neither is it possible to perform validation on HTML producing code fragments (i.e. only the entire generated output is validated). Also, it does not provide any clear information about the errors it has found; it only prints a message at the bottom of the page, stating that it has found invalid HTML.

For HTML validation in PHP there is a toolkit called TWINE [32], which stands for "Toolkit for Web-Interface Evaluation". It delivers a PHP class filter designed to submit HTML interfaces of dynamic web applications to the W3C HTML Markup Validation Service and then displays its results. The only code that has to be added to the PHP script is:

```
include "/path/to/HTMLValidator.class.php";
$v = new HTMLValidator();
$v->execute();
```

The advantage of this implementation is that it can quickly provide a detailed HTML validation report from one of the best public HTML validators available, namely the W3C validator [9]. But again, only the complete page output can be validated.

In contrast to Python, Perl and PHP, JavaScript is a client-side scripting language instead of a server-side one. This means that JavaScript is executed by the browser itself, instead of by an interpreter on the web server. This may bring some performance advantages, but makes it impossible to perform HTML validation before the output is printed on the browser, because the scripts are part of the output itself. Of course, such checking can be performed by adding a *One click validation* hyperlink to the document, that submits the entire HTML output to an external validator, such as the one provided by W3C. But again this can only

validate the whole document, and not fragments of it. And neither would it validate the HTML directly from the source code.

Another point is that Perl, Python and PHP scripts often generate JavaScript code themselves, instead of only HTML. This JavaScript code may contain errors and cause problems itself. This means that looking at the generated JavaScript code can solve some problems as well. However, the available HTML validators *do not* parse JavaScript, and some even throw an error message when they encounter any JavaScript code. In order to use these validators, the JavaScript code has to be wrapped in a `<![CDATA[ ... ]]>` tag, or be put in an external `.js` file.

4.3 Analysing the script source code

Most of the solutions described in Section 4.2 validate only the total HTML output of the script, and do not provide any clear information as to where any mistakes in the HTML output originate from in the actual source code of the script. However, performing such analyses can be tricky, but they are very useful: it would give a programmer more insight in where problems in his code may exist. In this section we discuss the advantages and disadvantages of HTML validation by analysing the script's source code.

A disadvantage of such an analysis is that while examining the code, we will also have to deal with the actual structure of the scripting language itself, and not just the HTML output. The validator can get misled by this, for example by commented code, or commented HTML strings that will not even be printed by the script at all. This can simply be removed from the source code (e.g. by parsing the code and then leaving out the comments), but it would quite complicate validation checks that use regular expressions. Also, some of the checked code could actually be dead (unused) code, but still produce malformed HTML. This may result in discovering errors about a certain piece of code (e.g. a procedure), while its results do not appear in the actual HTML output.

Another disadvantage is that we would have to keep track of the actual structure of the script itself; e.g. all procedures could be at the top of the document (which is common for most languages), but some languages allow functions to be defined at every possible position in the code (like in PHP), sometimes even inside functions itself. Performing a validation using regular expressions on such a source file could give a lot of false warnings, or miss actual warnings. This problem can be solved by using syntax-directed analysis instead, which is more complex but should give better results.

However, even syntax-directed analysis may miss some potential problems, for example an opening tag may be closed in some other procedure, but the syntax-directed analysis does not analyse this procedure. We consider a situation such as this undesirable: it is good practice to have functions open and close all the HTML tags they use, instead of having one function printing the opening tags and another the closing tags, because this would prevent many potential problems. These and other coding practices are discussed in Chapter 5.

Lastly, a major disadvantage is that such a validation may not recognize all tags that are actually printed. For example, the expression `"<" . "a" . " " . "href=\"test.html\"" . ">"` would lead to the text `<a href="test.html">` in the HTML output, but if we validate by source code, it could miss this tag easily, if the string concatenation is not evaluated. Even more obscure examples can be thought of, for example `"<" . foo($a) . ">"` where `foo($a)` gives a tag based on the value of `$a`. It will get even worse if some of the HTML output is read from user input or even an external file, of which the contents are even unknown at all to the script's source code. For situations such as these, a HTML validator that analyses the source can do nothing and only assume things. However, the source code can be written in such a way that the rest of the generated HTML is correct.

Advantages of HTML validation by analysing the source code is the fact that better feedback can be provided, on where validation errors originate from. Because we now know exactly in which string the tag is printed, the validator can refer to its exact location in the source code, instead of simply stating that the HTML is malformed, and for which tag. This would help solving HTML validation related problems a lot faster, because manually scanning of the entire source code to find the misused tag, is no longer required.

Common HTML validators halt validation when a (major) structural error in the HTML document has been detected (e.g. a missing closing tag). However when a closing tag is present, but not the correct one (yet it is similar to the correct tag), the HTML validator fails also. But, in such cases it could be rewarding to continue the validation, as if the correct tag was used[1]. Most HTML validators (including the W3C validator) do not

[1] Of course, we still report such situations to the programmer.

support this (because they use a DTD that only parses documents strictly), which may require validation of the document to use multiple passes, to find all errors. By allowing certain combinations of these tags, i.e. "loosely validate the source code", more problems can be detected in a single run of the validator. Of course any HTML validator can use this approach, but a validator using a DTD would be incapable of this.

A remaining issue are the JavaScripts. These could also be checked using this kind of validation, but then we would also require some understanding of the JavaScript language, and we even have to check the HTML generated by the JavaScript code. This could be tricky, because JavaScript can impact many locations of a HTML document[2], and using our method of HTML validation we may not have a clear overview of these, because some of the locations affected by the JavaScript, may not have been encountered yet. Lifting the JavaScript out of the code is out of the question also, because the JavaScripts can also print directly to the browser, and lifting these code fragments could deform the page layout. Therefore it is best to just ignore the JavaScript code.

Of course, HTML validation by analysis of the source code is not perfect, because some mistakes may be overlooked, and even some false positives may be found (especially with use of regular expressions). Still, a lot of problems can be successfully found, and more accurately reported than only validating the whole HTML output would do. Chapter 8 and 10 will illustrate this.

[2] JavaScript uses the HTML document's structure to refer to certain forms, and elements of these forms. Also, a JavaScript can change the appearance, and the value for such an element, triggered by events on the website itself.

Chapter 5

Coding practices and protocols

In Chapter 4 we discussed how HTML validation could help solve a lot of problems in a script. However, only doing HTML validation may not be enough. For example, web forms may have (hidden) variables, but the actual script never uses these. Detection of such variables can be quite complex, because it would require analysis of the entire script to find all the used variables. Another problem is to ensure that input variables in a GET context (obtained from a URL), and in a POST context (acquired from a submitted form of this, or another page) are handled in their appropriate context. This would require the programmer to have full knowledge of the use of all of these variables, because an analysis may not find all of them, due to external pages being involved.

Another problem is a function that prints a certain opening tag, with the corresponding closing tag being printed by another function. This is undesirable, since it can be tricky to find out where it may go wrong (e.g. if one of the functions is not always used, or executed in a conditional statement such as `if ... else`), when the HTML validator finds such an error.

Preventing problems such as these can be achieved by enforcing some - often simple - rules on a script, known as *coding practices*. Such rules can be quite simple to specify, such as demanding that *"a function should print all the closing tags of all the opening tags from HTML constructs it prints"*. They can also be more complex, input validation for example, in which all data obtained from files and web form fields, have to be processed by a validation function, before being allowed to be used in the actual script itself.

Another issue are databases. Many websites are supported by a database, and typically use SQL to query it. Using such a database requires the script to ensure that the database is really opened – and not a dead connection is used –, and that the queried tables really exist, as do their fields and everything else. Each scripting language has its own database interface, or several of them for different types of databases. Coding practices could be used here to ensure that a database is correctly opened and closed, and that correct table and field references are used.

Also other protocols can play a role in a script. For example, some languages support the notion of sessions. A session is an instance of a visitor to a website, and a session environment gives extra support to such an instance, e.g. by providing session variables, that can be used to store information, that can be shared between pages during this session. In essence, sessions add the notion of *state* to a website. Cookies and file handling are other examples in scripting languages that use a protocol. Coding practices can be applied for each protocol, to ensure that it is respected, and not misused.

In this chapter we discuss several patterns of coding practices, that can be used to ensure that a code fragment does not violate certain protocols (such as database handling), or produce erroneous output. This list of coding practices can hardly be a complete list of everything a programmer could consider for keeping in mind when writing his scripts, and some of the discussed items are even quite trivial. However, these coding practices can be quite useful in preventing a lot of the problems we discussed here, and are in general applicable to all of the scripting languages that we discussed in Chapter 2.

5.1 Input handling

Data obtained through web forms, local files and external sites cannot be trusted. Such data may contain malicious information, with the sole purpose of sabotaging a site. Therefore, all input has to be carefully filtered first, before being allowed to propagate through the program. Perl has a convenient security feature to handle such situations, the *tainted data* mode, and it throws exceptions when tainted data is allowed to propagate to a protected context, such as `system()` and `exec()` calls. Other languages, for example PHP, do not have a taint mode feature, but this can be simulated by adding a few functions that validate the tainted data: checks on every kind of user input, that confirm if these are processed by a validation function before being passed on in the program, could identify most security leaks. Such an *untaint* function should check all user input for tainted symbols like parentheses, semicolons and reserved keywords.

Another issue is how to treat each of the kinds of user input. For example, information obtained through a local file can be considered as less tainted, because it comes from the site itself, and so could have been untainted before it was written to that file. External files should then be considered as fully tainted because the contents may not be trustworthy, as their contents are typically unknown. Input from web forms through GET and POST variables are also fully tainted, because these may come from a malicious user, so they need to be carefully validated.

Some languages offer the possibility to consider GET and POST variables as the same kind of data. For example in PHP the `$_REQUEST` array contains all web input variables. However, it is not always desirable that a web form can be filled in by using the URL instead of the actual web form. So, use of the `$_GET` and `$_REQUEST` variables in PHP could be forbidden, or at least discouraged. However, a lot of information can also be passed on by other means than web forms, e.g. with session variables or cookies. Session variables are handled internally, so they can generally be treated as safe, but cookies are sent between browser and web server, so these can still contain tainted data.

We discuss some more specifics of tainted data in Section 5.3, where we discuss protocols for accessing databases and files.

5.2 Type inference related

When using full type-inference, a lot of typing mistakes can be detected, but type inference is very slow and expensive. Therefore, it is very useful to have some less expensive methods available, that can do some of the type inference work as well, at a lower cost. In this section we discuss a few.

Different languages often have different operators for string concatenation. For example, Java uses the `+` operator for both string concatenations and adding numeric values, but PHP uses the `.` for string concatenations instead. The same can be said for many other languages (e.g. Haskell uses `++` for list concatenations, which include strings). In statically typed languages a mistake such as this always leads to a type error, but in a dynamically typed language this is not the case. Here, a coercion is done to "transform" the found value into the expected type. Usually this proceeds without any problem, but there are a lot of cases where this goes wrong (e.g. a string to an integer coercion *can* fail), and then a *neutral* value is chosen (e.g. a non-numeric string converted to an integer would result in the value 0). Often this is unintended and would lead to unwanted results for a calculation.

Some languages allow assignment to variables within expressions. The result of such an assignment expression is the value that is assigned to this variable. Often this is used in harmless multiple assignment expressions, such as `$a = $b = $c = 5`, but in cases such as `$x = ($y = $z) + ($y++)`, it becomes quite impossible to determine which value is assigned to where. Strictly spoken, `$y++` is an assignment also, and depending on the language, these are executed before, after or during the statement they appear in. Therefore, it is good practice to make rules such as *"no expressions of the form* `$a++` *inside assignment statements"*, and *"no nested assignment expressions in expressions"*.

Many programmers use assignments in conditions of statements. An example:

```
while ($row = mysql_fetch_row($result)) { ... }
```

This PHP example can be used for the retrieval of data from a MySQL database query, row by row. But, the type of such assignments in conditions can also be other than *boolean*, in which case unwanted coercions take

place. Such assignments could be prohibited, or only a small set of functions could be allowed, i.e. all other functions (and expressions) are refused, and reported.

5.3 Protocols

In this section we describe some of the most used protocols in scripting languages, and how we can ensure they are correctly used, by applying some coding practices.

Databases

All scripting languages have their own database interfaces, often even multiple ones, to support each kind of database. For example, PHP offers support for MySQL, PostgreSQL and several other ones. First, we discuss how the different scripting languages that we discussed in Chapter 2 handle database connections.

For Perl, a module called DBI [5] is used to handle database protocols. It is independent of the type of database that is actually used, i.e. it can work with every kind of database (whether it uses SQL or perhaps even something else for querying). The convenience of this method is that the programmer does not need to have any knowledge about the details of the actual database being used, i.e. he can write queries that often still work if another type of database is used. However, it is still required that the programmer checks if the queries are correct; the programmer is still required to know all the fields and tables of his database.

Python does not have the convenience of a universal database interface; it does however offer an API [20] to make your own database interfaces, for which several (MySQL for example) already exist. The same problems as with the Perl solution exist as well: the programmer still requires knowledge of the database structure (tables and fields) in order to ensure the correctness of his queries.

In PHP there is no universal database interface either. Instead, several modules offering their functionality in the form of functions, classes and constants are available for each database type, each with their own prefix (e.g. the PHP MySQL functions start with `mysql_`, and the PostgreSQL functions with `pg_`). To use any of these databases however, the corresponding module needs to be installed in the web server first.

Because JavaScript is a client-side language, all of its code is included in the HTML output. Therefore, no database interfaces for JavaScript are known to exist, due to the large security risk. For example, to access a database, its server location and its password information are required, and it is *very* unsafe to have these clearly visible in the HTML output document.

What all database interfaces have in common, are the tasks that need to be performed in order to have a safe and reliable access to a database server. A connection has to be opened at the start of the page before any queries can be performed, and at the end, the connection must be closed. Simple checks could be put in place to ensure that this is the case for every page. Also, it is good and common practice to make a function that takes care of the database connection, instead of making the connection directly in each script file. This not only saves a lot of work when the connection details have changed, it adds some extra security – the include file could be moved to a directory which is not accessible by web page visitors directly, so that the connection details are never compromised – as well.

Other coding practices that can be useful here are applied on the actual queries itself. For example, when accessing a database table using SQL, these queries are often composed using variables of the script. These variables usually contain field values, but they can also be used in the place of field names, or even table names. This could be very unsafe because these variables would have to be validated, to ensure these field and table names really exist. Thus, disallowing such occurrences is a good coding practice. Also, the field and table names that are used, could be matched using a reference table, to check if the queries itself are actually correct, before the script is even executed at all. However, this would require the programmer to have a reference table of the available tables in the database, and their fields. Another coding practice that is useful, is one that checks results of queries are actually processed (and, that its results are actually processed *after* executing the query).

Sessions

Some scripting languages allow the notion of state in websites, by the use of session variables. These variables can store any value (even whole data structures), which are saved in a session file which is shared between all pages for that specific session instance. This can be very useful, but also very unsafe: session variables can also be assigned tainted web input. So, before a value can be assigned to a session variable, it has to be untainted first.

Also, when using sessions, they need to be properly started at the beginning of a page that uses them. In PHP the function `session_start()` takes care of this task. However, PHP does not check if this function is called before session variables (or related functionality) are used, i.e. some checks should be done to ensure that sessions are correctly opened, before their functionality is used. It does however check if something is printed to the browser, before a session is started. The closing of sessions should only be done when the session is no longer necessary, i.e. when the user logs out from the website. Sessions can also be set to expire automatically, e.g. three hours after creation.

Files

The handling of files is similar to that of sessions and databases, i.e. they first have to be opened, then read from or written to, and then closed. However, a distinction has to be made to whether the file is read locally (i.e. from the server itself), which can be trusted more easily, or remotely, e.g. from a socket, or from a website located elsewhere. Remote files are likely to be more tainted than local ones, so these need to be checked more strictly.

Also, the mode of the file handle (read or write) needs to be considered, before doing any operation on a file. For example, a file that is opened to be read-only, may not be written to. Therefore all file accesses need to be carefully analysed to determine which operations are valid, and which are not.

Other

More protocols can be thought of, but they typically follow the same order of operations: opening and initialization of some connection, followed by several operations that have to be performed on some data, and finally a closing procedure. A simple generic check can be used on all such protocols, with some additional checks added for the protocol-specific features.

5.4 Coding styles

No programmer writes his code in the same style, or uses the same methods to write the same piece of functionality. In this section we describe some coding practices that could make source code more readable.

Print statements should not be scattered throughout the script. It is better to collect the entire document behind the scenes with strings, and then print it using a single statement. This will ensure that everything can be validated before it is sent to the browser, i.e. erroneous parts can be left out of the site entirely, or simply be replaced with a nice error message. If such HTML validation is done for each section of the site, before it is added to the variable collecting the contents of the site, full validation can be done behind the scenes, by either applying it on the entire string, or the generated substrings. However, this would require that the programmer knows which strings are going to be printed and which are not. Also when validating such strings, it is required to know if these strings contain printable HTML, or other data.

Another problem when printing data, is the fact that a scripting language may have more than one means of printing text to a browser. For example, PHP has an `echo` statement, but also a `print` and a `printf` function, both also designed to output data, each with their own specific functionality[1]. Using the `echo` statement, several strings can be printed directly, while `print` only allows one string. The `printf` function also allows only one string, but formats the output using a given format-string. The difference between `print` and `echo` is that `echo` is a language construct and not a function. Generally it is not a problem to mix the usage of

[1] PHP has even more printing functions. For example, `print_r()` prints a given text in human-readable format. This makes e.g. arrays more readable. Please see [24] for more information on such functions.

`print` and `echo`, but to keep scripts readable and tidy, it is better to just use either one of them, and not both. Also, the function `printf` is rarely used.

Another printing issue is the notion of strings. For example, PHP has two kinds of strings, single-quoted, and double-quoted. The difference is that single-quoted strings are printed as-is to the browser, while double-quoted strings may contain variable occurrences (e.g. `<b>$a</b>`), which are interpolated (i.e. the value of these variables are inserted in the string) before they further propagate through the script. Because double-quoted strings have all the functionality that single-quoted strings also have, the use of single-quoted strings can be disallowed, which means that only one style should be used. However if the actual purpose of the single quoted string is to display the `$` character (i.e. no interpolation), it can also be achieved by escaping it in a double-quoted string (e.g.`"\$test"`), which gives the same result.

PHP allows function and class declarations inside `if-then-else` expressions, which may result in confusion about what the function or class will actually be after the execution of this statement. Such situations may have significant performance advantages, if the `if-then-else` is executed once for declaration of the function or class. However, this reduces the understandability and the readability of the code, and makes analysis much more complex. Therefore, it is good practice to disallow the declaration of new functions and classes within inner code blocks.

Large code blocks are often source of possible problems, and are usually quite unreadable. Splitting these in smaller, more readable blocks, e.g. by putting these into functions or class methods, increases the readability and understandability of the code. Also, this can help the HTML validation somewhat: by ensuring that these smaller blocks open and close the specific tags they use, it can be ensured that closing tags are never missed.

It is good practice to define what an include file is allowed to do. For example, we could disallow these to print directly to the browser (i.e. disallow all printing functions and constructs), or allow these to only contain functions, such that no statement is executed when the file is included. It is also practical to look for situations where include files include other files, and to try to detect cycles here. Situations like these may lead to a never ending recursive loop of files including each other.

In the PHP language there are variables, functions and constants for which new names have been introduced. However, their old equivalents are still allowed, to be backwards compatible. However, it is not ensured that these deprecated names still exist in a later version, so their use should be strongly discouraged. Simple checks can be done to identify such deprecated identifiers, and to replace them if possible. An example of such a deprecated name is the `$HTTP_POST_VARS`, for which the name `$_POST` can be used since PHP version 4.1.0.

In languages using global variables (i.e. variables that are defined outside of a function, but accessible inside these functions) it is possible to use the same variable names for global and local variables. However, it gets confusing when a variable name that is used in many functions as a global variable, gets used in a function that does not use this specific global variable, but has a local variable with the exact same name. Also, if this global variable is eventually needed in this function, all the instances of the local variable with this name in that function need to be renamed, to avoid name clashes, and loss of data. Therefore, it is good practice to never give local variables the same name as common global variables.

5.5 Conclusion

A lot more coding practices can be thought of; the collection we discussed in this chapter is hardly complete. However, it gives a good overview of what situations might arise when using a (dynamically typed) scripting language. In the next chapters, we will consider these problems for one of these scripting languages, i.e. PHP.

First we will discuss the specific features of PHP (Chapter 6), next the type inference system (Chapter 7), and finally we discuss several analyses that will handle the problems discussed in this chapter (Chapter 8). Then, we discuss an implementation of a framework and tool that applies the analyses that we discuss in these chapters (Chapter 9 and 10).

Chapter 6

PHP

PHP is often considered an easy language to construct websites with, due to its vast community, support and immense function library. It is also known for its dynamic nature, both in language structure and constructs, as well as in variable scoping and typing and function handling. However, this dynamic nature of the language unfortunately makes it very hard to analyse, and very easy to make mistakes. These can range from very simple ones, like unintended assignments in conditions, to larger ones (e.g. tainted data giving security issues) that can even cover several source files at the same time. In this chapter we give an overview of the PHP language and its features.

Please note that the goal of this chapter is not to give a complete specification of PHP, but to give a clear overview of what the features of this language are, and how they work. A full description of the entire language and its features can be found on [24]. The first section of this chapter is an overview of the language's features that we consider in our research, then we give a list of restrictions, i.e. features we will not include in our analyses, and finally an overview of the program flow in PHP, which is required for the next chapters, where we define analyses on PHP.

6.1 Language features

Programs in PHP can be seen as a list of statements, separated by semicolons, optionally complemented with a list of functions and class definitions (Figure 6.1). There are several kinds of statements, each with their own use and purpose. In this section we list the language features.

6.1.1 Variables, constants and expressions

Variable names must start with a `$`, followed by an identifier, which consists of alphanumerical characters and the underscore. Constants are atomic values, and include strings and composite constant values like arrays and objects. With regards to associativity and priority, expressions are analogous to languages like Perl and Java. Sub-expressions are always evaluated before the actual expression itself. For example, consider the expression `$a + $b * 5`. First `$a` is evaluated, and then the expression `$b * 5`, of which the `$b` and `5` are evaluated before the multiplication. After evaluating the multiplication expression, the `+` operator is considered, completing this expression.

Two special expression constructions are the `array` and `count` constructs, which create an array, and give the size of the array (or any other expression) respectively. When used on non-array expressions, `count` will always yield 1 as result, or 0 if used on an undefined variable.

Type casting can be done to *ensure* a value is of a given type. However, in general this has little to no effect at all on the value, except if it is of a distinctively different type than the programmer intended it to be (e.g. object to string).

constant ::= *int* | *float* | *string* | *boolean* | `null` | *array* | *object*
variable ::= `$`*identifier* (`[` *index* `]`)*
index ::= *string* | *int* |*variable*

exp ::= *constant*
| *variable*
| *exp* `=` *exp*
| *exp binop exp*
| *variable* (`++` | `--`)|(`++` | `--`) *variable*
| `(` *type* `)` *exp*
| *unop exp*
| `(` *exp* `)`
| *identifier*`(` *exp** `)`
| *exp* `?` *exp* `:` *exp*
| `array(` *exp** `)`
| `count(` *exp* `)`

binop ::= `+` | `-` | `/` | `*` | `%` | `.` | `==` | `!=` | `>` | `<` | `>=` | `<=` | `&&` | `||`
unop ::= `-` | `!`

statement ::= *assign_statement*
| `echo` *exp**
| (`include` | `include_once` | `require` | `require_once`) *exp*
| `if (`*exp*`)` *statement* [`else` *statement*]
| `{` *statement** `}`
| `break`
| `continue`
| `return` [*exp*]
| *variable* (`++` | `--`)|(`++` | `--`) *variable*
| `for (`*decls* `;` *exp* `;` *incs*`)` *statement*
| `foreach (`*exp* `as` [*var* `=>`] *var* `)` *statement*
| `do` *statement* `while (`*exp*`)`
| `while (`*exp*`)` *statement*
| `switch (`*var*`) {` *case** `}`
| (*assign_statement*|*call_statement*) `or` (*assign_statement*|*call_statement*)
| *call_statement*

assign_statement ::= *variable* (`=` | `+=` | `-=` | `*=` | `/=` | `%=` | `.=` | `&=` | `|=` | `^=` | `~=`) *exp*
call_statement ::= *identifier*`(` *exp** `)`

case ::= `case` *constant* `:` *statement*
| `default` `:` *statement*

function ::= *identifier* `(` (*variable* [`=` *constant*])* `)` `{` *function_statement** `}`
function_statement ::= `global` *variable*
| `static` *variable* `=` *exp*
| *statement*

program ::= (*function* | *statement*)*

Figure 6.1: Specification of statements, expressions, identifiers and constants in PHP

6.1.2 Statements

There are several kinds of statements. In this section only the statements not involving branching and loops are discussed. PHP has assignment statements like any other imperative language, including the ability to shorten certain sequences, e.g. `$a = $a + 3` can be shortened to `$a += 3`. In the case of `$a = $a + 1` it can even be shortened further: `$a++`. Expressions such as `$a++` and `$a--` can also be used in expressions, and are either executed before the entire expression itself (`++$a`), or right after (`$a++`). Assignments can appear inside expressions, but they may not be of the shortened form (i.e. `$a += 2` here. In that case a parser error is thrown.

Printing text to the browser is performed by use of the `print` function (or any other function with text printing capabilities), and by the `echo` statement. The advantage of using the `print` function is that it can be used inside expressions as well. `echo` takes one or more expressions as arguments, and prints them as strings. There is also a `print` statement, but it is simply an alias of `echo`[1]. Either case, it is often considered good practice not to mix the usage of `print` and `echo`, if only to keep the scripting code tidy.

It is also possible to group statements into *blocks* or compound statements, by surrounding them with curly brackets. This is especially useful when using control structures and loops.

6.1.3 Choice structures and loops

Choice structures in PHP include the `if ... else` conditional statement, the `... or ...` statement, and the `switch` construct which contains `case` statements.

The `if ... else` structure works analogous to that in Perl and Java. After evaluation of its condition, either the *then* branch is executed (if the condition is `true`), or the *else* branch, in case the condition fails, and such a branch is present. Both branches can be single statements, or compound statements surrounded by curly brackets.

The `... or ...` construct is quite different. The left branch is executed first, and only if it causes a (fatal) error, the right branch is executed as well. If no error occurs during execution of the left branch, the right branch is never executed. The only allowed statements in an `... or ...` statement are assignments and function calls. There also exist a `... and ...` statement, which requires both branches to succeed, and a `... xor ...` statement, but these are rarely used. The `... or ...` statement is often seen in situations such as database calls, where the right branch is often a call to the `die()` function, which halts execution, and shows an (user-defined) error message.

The `switch` construct can be seen as a more general case of `if ... else`. Given an expression (which can be a variable or function call), a series of statements is executed, depending on the value of this expression. This sequence of statements usually ends with a `break` statement to mark the end of a code block. Each case is covered by a `case` statement holding a value for the expression, and for remaining cases (for which no value is specified), the `default` keyword is used. It is possible to remove the ending `break` statement for any of the cases, but this is rather uncommon and strongly discouraged. Case statements can be 'stacked' as well, in cases that one code block is used for several `case` values.

Loops in PHP come in four forms, each with their own use. The `for` loop is used to iterate by incrementing one or more variables (usually integers). These can then be used e.g. to iterate through an array with numeric indexes, also known as a non-associative array. Such arrays however can also be iterated with a `foreach` loop, which can also handle associative arrays, which have strings as keys, and are often used as key-value maps.

For loops that do not require iteration through a given data structure, the `while` and `do ... while` loops can be used, which operate on a single condition that has to be met to stay inside the loop. The difference between `while` and `do ... while` is that a `do ... while` loop executes its body at least once; the condition is only tested after executing the body here. This kind of loop is very similar to the `repeat ... until` construct known from Pascal-like languages.

6.1.4 Includes

Include statements allow you to include all the code, including all function and class definitions, from one file, into another. This can be very useful when the programmer needs to divide his code into modules, for

[1] In fact, there are both `print` and `echo` statements, *and* functions. And both are aliases.

example when putting all database handling statements and methods in a separate file.

There are two main types of include files: `include` and `require` files. The difference between the two is that if an `include` file is missing, a warning message is generated, but execution continues normally, as much as possible. In the case of missing `require` files however, execution is halted. You can also demand that a file can only be included once (e.g. at the top or bottom) in a PHP script. In this case, the `include_once` or `require_once` variants are used.

6.1.5 Functions

Like in all languages that support procedures or functions, in PHP functions make it possible to add another level of modularity to your code, aside of includes, and allow code to become more readable as well. Functions cannot directly access any variables that have been declared outside of it. However, it is possible to declare one or more variables in the function header, which can be used to pass information from outside into the function.

Parameters are passed by value, and there is only one (nameless) result parameter, which can get its value from any `return` statement in the function. Recursive calls of functions are allowed. Functions may not be used as parameters. Even though [24] states that there are *callback* parameters, which contain the name of the function to be called, these are actually *strings*. This may look like higher-order functions in PHP, but we are unable to handle these callback parameters, because they are passed to the functions as strings. There is also a notion of *default parameters*, which means that these specific parameters may be left out of the actual function call, and get a *default* value when this is the case. Only the last parameters may be default parameters, otherwise a parser error is thrown.

Returning the result of a function is done with the `return` statement, for which the expression is optional. This `return` statement may also appear outside of functions, where it causes a jump to the end of the file, ending its execution.

The order of function definitions is irrelevant. They can be defined anywhere in a program block (e.g. a compound statement, or an include file, even inside functions themselves), and these functions can then be called from anywhere from the scope on which they are defined.

6.1.6 Scoping

In PHP there are two scopes: the global scope, which is the scope for all variables not used in functions, and the function scope, which is a local scope, for *each* function, that is completely separate from the global scope. This means, that in a function you cannot normally access any variable declared outside of that function. The only way to pass information to the function is by using the function parameters.

Fortunately, it is possible to manipulate these scoping rules. By using *global* and *static* variables, introduced by the `global` and `static` statements, the scope of a single variable can be altered to either of these scopes. The two statements change the scope of a variable as follows:

- `global` declares a variable as global. This means, that from the program point where the variable is declared as global, its value is taken from the global scope of the script. All changes to this variable are also saved on the global scope. The variable's value before the declaration to global scope will be lost.
- `static` declares a variable as being static, causing this variable's value to be retained when program execution leaves the function. When declaring a variable as static, an initial value must also be defined, which will be placed on the function's static scope. While global variables are retained when leaving the function and are also accessible *outside* of this function[2], static variables are only retained for use within the body of the function where they were declared. Again, the old value before declaration to static scope will be lost, if the variable was a local variable.

Variables can be declared as `static` or `global` anywhere inside a function. The use of this feature is strongly discouraged however. A variable can even be static at first, and then later become global, and then static again. This is also strongly discouraged, because it adds unnecessary complexity to the function and it gets

[2]However they are not accessible in other functions, even those that are called from inside this function, unless it is declared as global there also.

confusing to find to which scope that variable belongs. It is common and best practice to put all static and global declarations at the top of the function, to avoid confusion.

Superglobals

A variable can be declared as `global` outside of a function, which will turn it into a superglobal. Such a variable is accessible *everywhere* in the script, even inside every function. There are several predefined superglobals, the most important of which are `$_GET`, `$_POST`, `$_REQUEST`, `$_SESSION` and `$GLOBALS`. The first four of these handle GET and POST variables which are given by the web server, and session information. `$GLOBALS` however is an interesting one: using this superglobal you can access any variable on the global scope in any function, as if these variables are declared as global in these functions themselves. Often this can be used as an alternative to the `global` statement, or in situations where accessing a variable with the same name in both global and function scope, without losing its value through global declaration, is required.

It is also possible to automatically turn *every* variable in PHP automatically into a superglobal, by means of the PHP configuration directive `autoglobals`, but use of this is *strongly* discouraged due to the severe security risks this will introduce.

6.1.7 Coercions and converted values

In PHP, every primitive value is stored as a string. These values are only translated to values of other types when the execution needs a value of such a type, for example in a numeric calculation, or in a conditional (boolean) expression. However, sometimes a string value *cannot* be turned into a numeral. In these cases, a warning is generated (though these are generally hidden), and a neutral value, e.g. `0` in the case of a number, is used instead. In Figure 6.2, a list of all common coercions in PHP is listed, with their result value and the value in case the coercion fails.

Generally, only coercions between strings and integers, floats or booleans should occur – because these are meaningful –, and not between other types (like arrays or objects). Strictly spoken these are allowed too (e.g. making an array from a single value simply by casting it to an array, which is actually very useful), but typically they have little meaning, and in some cases they destroy information (e.g. in the array to string coercion). Typecasts from the resource type to any other type (and vice versa) generate an error message in most PHP versions, but in some versions it is allowed to convert a resource type value to a string (i.e. in version 5.0 and later).

6.1.8 Other language constructs

There are also several other language constructs in PHP, most of which are in experimental stage, or are rarely used. We discuss a few of these here.

Like in Java, PHP can use a `try ... catch` construct to trap errors with, much similar to the `... or ...` construct. However, `try ... catch` is more robust and does not produce the warnings that may appear when one or both of the statements in `... or ...` go wrong. Also, `try ... catch` can be used on a group of statements, instead of just single assignments and function calls as with `... or ...`. So in general, this can be seen as a generalization of `... or ...`.

In PHP 6, a new construct called `comefrom` is introduced, which allows the programmer to insert pieces of code in one or more places of his PHP script. This can be very unsafe (and no doubt there will be plenty of unsafe examples of this construct), but it can also be very useful, e.g. when print messages are used throughout the script for debugging. With this `comefrom` construct they can all be placed in one single include file, which makes editing (or removing) them a lot easier. So, this `comefrom` construct makes it possible to do some form of aspect-oriented programming [34].

6.2 Restrictions

Analysing all of PHP's features, abnormalities and possibilities was not possible in the time frame for this project. However, we have tried to include as much of the language's features as possible, that we considered to be interesting enough for this project. We decided on the following restrictions:

From	To	Result value
array	string	`"Array"` (*strongly* discouraged)
object	string	`"Object"` (*strongly* discouraged)
string	int	`0` (in case of failure), otherwise the string as an int value
string	float	`0.0` (in case of failure), otherwise the string as a float value
int (e.g. `4`)	string	The int value as a string (e.g. `"4"`);
float (e.g. `2.5`)	string	The float value as a string (e.g. `"2.5"`);
int (e.g. `4`)	float	No change (`4.0` in this example)
float (e.g. `1.5`)	int	The float value truncated (in the example, `1`)
int or float	bool	`true` when the int or float value is non-zero, `false` when int or float value is zero
bool	int or float	`1` when bool value is `true`, `0` when bool value is `false`
bool	string	`"1"` when bool value is `true`, `""` when bool value is `false`
string	bool	`false` for `""`, `true` otherwise
any type	array	An array containing only the given value
any type except object	object	A null object reference (*strongly* discouraged)
object	int or float	`0` (*strongly* discouraged)
array	int or float	`1` (*strongly* discouraged)
array	bool	`true` (*strongly* discouraged)
object	bool	`false` (*strongly* discouraged)
resource	any non-resource type	Error message (version 5.0 or later)
any non-resource type	resource	Error message (version 5.0 or later)

Figure 6.2: Coercions and converted values in PHP

- Class definitions, and language constructions using classes and objects are not considered in our analyses. Having support for classes would add little extra scientific value for this project, and implementing the support for them would cost a lot of time and work. Even though they are quite often used in professional libraries such as PEAR [23], the average website programmer does not seem to use them very often.
- No usage of the `try ... catch` construct. Even though this would not be very difficult to implement, it would not add much value to our research, because we already support a similar construct, namely the `... or ...` statement, which is used more often than `try ... catch`. Also, `try ... catch` makes uses of classes, which we do not support.
- No superglobal declarations. Declaring these manually would complicate the scoping rules, because it would make it necessary to keep constantly track of which variables are declared as superglobal, and where this happened. Global variables may still be used, but they may only be declared in functions. Also, the system predefined superglobals may still be used as these do not give these complications.
- Global declarations are assumed to be at the beginning of the function, followed by the static variable declarations, and finally the rest of the function's statements. In PHP it is possible to declare a variable as static or global at any point, making it also possible to switch between these scopes. However, it would make analysis on these global and static variables a lot more complicated, if this scope switching is allowed.
- Each `case` statement inside a `switch` construct must end with a `break` statement. Also, `case` statements may not be stacked in case the same code block is used for several case values. We chose to do this to reduce the complexity of our parser and abstract syntax tree.
- Functions may only be declared on top level, outside of any language construct (e.g. inside an `if ... else` that defines the same function but with a different definition in both branches), and outside of any other function (i.e. no inner functions). In other words: they appear on the same level as top-level statements. However, we do allow functions to appear inside of include functions, and between top-level statements.
- No support for "obscure language features". With this we mean that e.g. assignments are of the form

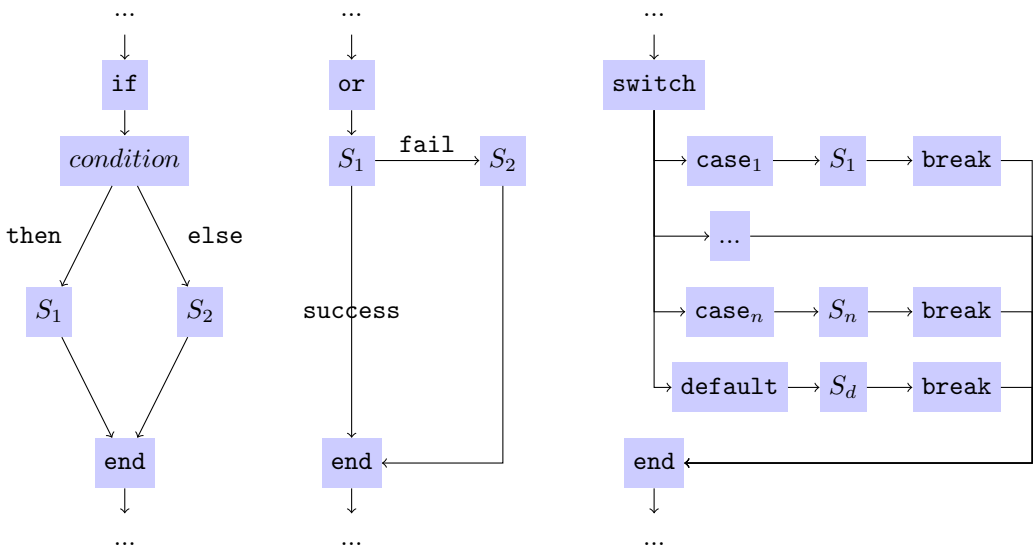


Figure 6.3: An overview of the flow for control structures in PHP

variable := *expression*, where the PHP compiler actually supports *expression* := *expression*, where the expression before the := can be any expression. Such obscure features only clutter the language, and makes analysing PHP extremely difficult. This also includes the use of HereDoc expressions [26], which can be easily converted to double-quoted strings.

6.3 Program flow

For a proper understanding of how PHP programs are executed, an overview of their flow is presented. Most of the time, statements are executed in linear order, but at some points branching occurs (e.g. in control structures like `if ... else`), or loops are used. In Figure 6.3 the flow for control structures is displayed.

The three control structures displayed, namely `if ... else`, `... or ...` and the `switch` construct all have different paths of execution. Each arrow is a step in the execution path of the structure. In the `if ... else` construct, one of the two paths is chosen (or none, if the `else` branch is missing), depending on the result of the condition. In the `... or ...` the left path is first executed, and if no errors occur, the right path is omitted. However, if the left path raises an error, the right path has to be executed as well.

The `switch` construct works quite like the `if ... else`, except that it accepts a single expression, and then it branches on (some of) the possible values of that variable, the cases. Each of these cases then represents a single branch, after which the program execution jumps to the end of the `switch` construct. Each case ends with a `break` statement, but in PHP it is allowed to omit this, to allow fall-through in this construct. However, our restrictions (Section 6.2) forbid this: all branches must have a `break` statement leading to the end of the `switch` statement.

In Figure 6.4, a representation of PHP's four different loop structures is given. Each structure has its own specific start and end node (i.e. in a `while` loop these are called *while* and *end-while* respectively), and its own path through its children nodes.

A `while` loop starts by evaluation its condition; a `do ... while` loop first executes the inner statement, and only then evaluates the condition. Therefore, a `do ... while` loop executes its inner statement at least once, while this does not necessarily hold for a normal `while` loop.

A `for` loop iterates a statement over one or more (typically numerical) variables, as long as at least one of the conditions is valid[3]. Usually a `for` loop iterates over a numerical variable between a lower and an

[3]Only *one* of the conditions of a `for` loop has to be valid. In essence, you could chain them all together with the boolean ||

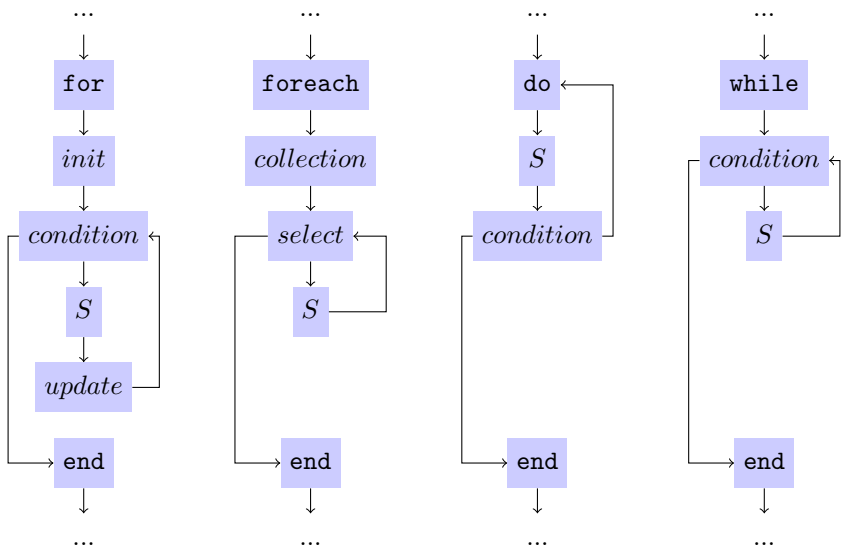


Figure 6.4: Flow of loops

upper bound. The `for` header consists of three consecutive parts: the declaration/initialization of the counter variables, then the condition, and finally the statements to update the counters.

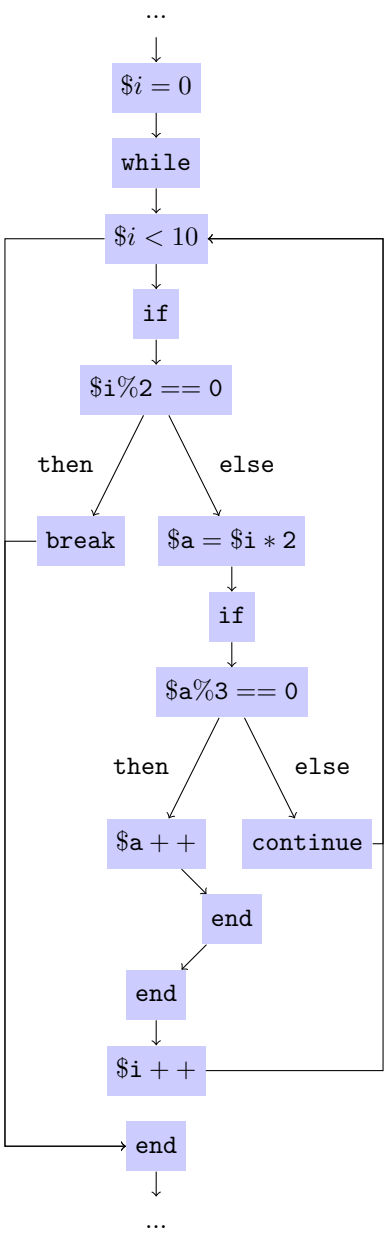
A `foreach` loop iterates over a collection, typically an array structure. There is no condition here: iteration continues if there are still values in the collection to consider.

There are two statements that can alter the flow of a loop: `break`, which makes execution leave the loop structure, and `continue`, which starts the next cycle of the loop[4]. Figure 6.5 illustrates this behaviour.

The `end` nodes in Figures 6.3, 6.4 and 6.5 are not physical commands or structures in the PHP code; these are artificial nodes that have been added to simplify the program-flow, and allows proper merging of information at the end of control structures and loops.

operator.

[4]For the `for` loop, the increment statements are executed. For the `foreach` loop, the next value from the collection variable is taken.



PHP code:

```
...
$i = 0;
while ($i < 10) {
  if ($i%2 == 0) break; else {
    $a = $i * 2;
    if ($a %3 == 0) $a++; else continue;
  }
  $i++;
}
...
```

Figure 6.5: The use of `break` and `continue` statements in a *while* loop

Chapter 7

Type inference for PHP

To solve the problems of type coercions which may lead to unintended values for functions and variables, we propose a type inference system for PHP. However, due to the extremely dynamic nature of the language, we try to keep our type inference system as close to PHP's dynamic nature as possible. This has been done by allowing variables to have *union types*, similar to [7, 2].

7.1 Type language

In PHP, all primitive variables are internally represented as strings[1], and converted to their required types through coercions when required. This makes it possible for every value to change to any possible type without any complications. However, in some cases, this may result in unintended behaviour, as we explained in Section 6.1.7, when the type to convert to does not have a valid value for this coercion.

To help solve this problem we define a type language that allows every variable to have any possible type, and to change to any other type needed, and ensures that no complications remain uncaught (like suspicious coercions). In Figure 7.1 we define the type language. Because we only work with concrete types – our type language does not support type polymorphism – type variables cannot be introduced.

A *type* in our type language is either a *base_type* or a $function\_type$. A *base_type* is either of the primitive types, such as *int*, $float$ or *bool*, or an *array* of any of these types. These primitive types are then used in a *type_set*, which can be any possible set of base types. A function type is a list of argument types (which are all type sets), with a result type set. This result type set is either a type set itself, or can be *void*, if the function has no return type.

Our type system is stricter than PHP's internal representation of values, because it says that `10` is a value of type *int*, and `"10"` a value of type *string*, while in PHP both values are treated equally. The information about a variable's type that we obtain, can then be used to detect coercions and eventually also to detect if variables may have multiple types at a single point in the script.

[1] Objects are an exception to this rule, but we do not consider these in our system. Resources are also an exception, but before PHP 4, these were represented as strings as well.

$type ::= type\_set \mid (function\_type)*$

$base\_type ::= int \mid float \mid bool \mid string \mid resource \mid object \mid array[base\_type]$
$type\_set ::= \mathbb{P}(base\_type) \mid \bot \mid any$

$function\_type ::= (type\_set)^* \rightarrow result\_type$
$result\_type ::= type\_set \mid void$

Figure 7.1: Type language for PHP

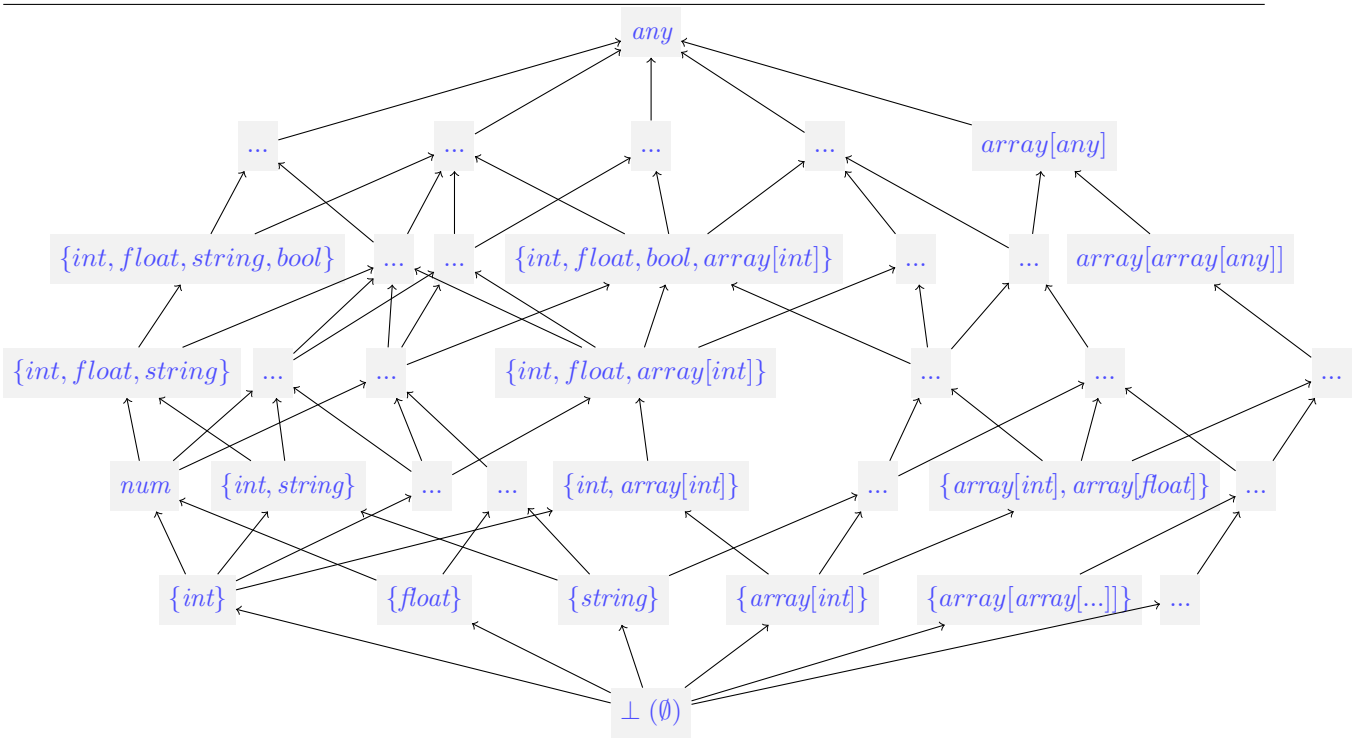


Figure 7.2: The (partial) lattice of our type language for PHP

We consider the type of an identifier to be a *type_set* in our type system. Preferably, these type sets contain only a single type, but during analysis this set can grow to a set of any possible size. In Figure 7.2 a lattice structure for a finite part of our type language is depicted. Starting from an empty set of types (represented as $\bot$), a variable v is assigned a *singleton* type of any kind (be it *int*, *float*, or even array types like $array[int]$ or nested array types like $array[array[int]]$). Each step in our lattice leads to a larger set of types, eventually reaching *any*, which is the set of all possible types[2]. Also, because the combination $\{int, float\}$ occurs very often, and both types represent a numeric value, we introduce a different notation, namely *num*, for this specific type set. Similarly, we use $array[any]$ as denotation for the list of all possible array types, and $array[array[any]]$ for all arrays of arrays, etc.

7.2 Type unification

During the type analysis, the type set of each variable can grow to arbitrarily large sets. Because it cannot be guaranteed that these type sets eventually stabilize at some point, nor that termination of adding new types would yield a precise result, another way of approximating the actual type set for a variable must be considered. After joining the type set with the newly calculated types, it should be replaced with a new type set, of which it is known that it will lead to stabilization, yet is still a safe approximation of the actual type set. This is called *widening* [39].

To achieve these stabilized type sets for a variable, the easiest solution would be to proceed to the maximum value (*any*) directly, but this would destroy any specific information, and this is not necessary in some situations. For example, when all types are arrays (or arrays of arrays, etc.), a more sensible solution is possible: $\{int, bool, string\}$ will be widened to *any*, but for $\{array[int], array[bool], array[array[int]]\}$, it makes more sense if the set is widened to $array[any]$ instead of *any*, because then the information that this set only contains array types can be retained.

Our widening operator ∇_k takes two arguments, ts_1 and ts_2, where ts_1 is the original type set, ts_2 the type set

[2] *any* can also be seen as the "type" of a variable if we are not exactly sure of its real, actual type. This is analogous to the Error data type in strongly typed languages, such as Haskell, when the compiler is unable to figure out the type of a function or data structure.

$\{int\} \nabla_4 \{float\} = num$
$\{int, bool, string, array[int]\} \nabla_4 \{array[string]\} = any$
$\{array[int], array[bool], array[string], array[array[int]]\} \nabla_4 \{array[array[string]]\} = array[any]$

Figure 7.3: Examples of type unification

we wish ts_1 to unify with, and k an integer number that defines how many types may appear in the resulting type set *at most*.

Our widening operator ∇_k is defined as:

$$ts_1 \nabla_k ts_2 = \begin{cases} \bot & \text{if } ts_1 = ts_2 = \bot, \\ ts_1 \cup ts_2 & \text{if } |ts_1| + |ts_2| \leq k, \\ array^i[any] & \text{if } |ts_1| + |ts_2| > k, \text{ and} \\ & ts_1 \cup ts_2 \text{ has only types with an array depth of at least } i \in \{0...n\}. \end{cases}$$

The use of ∇_4 is illustrated in Figure 7.3. With *an array depth of at least i* we mean: all types must be of the array type, where i is the actual depth of array nesting: $i = 0$ allows the types in our typeset to be of any type, $i = 1$ requires all types in the type set to be of the form $array[\tau]$, $i = 2$ requires all types in our type set to be of the form $array[array[\tau]]$, and so on, where $\tau \in base\_type$ (as introduced in Figure 7.1).

In fact any widening operator ∇ could be used in our type system, but the one we describe here works very well, as it still preserves some information about possible array types in our type sets, and not replaces it with *any* immediately. We choose to use ∇_k because it preserves the array information as much as possible.

Function types cannot be unified in this way, because their result types depend on the argument types (which are type sets themselves), and any difference in these argument types can lead to a different result type. Global and static variables have a similar influence. We cannot see from outside a function that it contains global and static variable declarations, and their impact on the result type. However, if all the argument types – and the types of the global and static variables – are exactly the same for two calls, then the result type is also the same for both calls[3]. More research is required to confirm the validity of this claim.

7.3 Type constraints

We define our type rules using the type language defined in Figure 7.1. We have chosen for a constraint-based approach. For each operation, e.g. a statement (like an assignment), an expression (function calls, operator calls), or a choice construct (like conditional statements such as `if ... else`) or loop (like `while`), a set of constraints is generated, that represents the operation's behaviour on types of variables and functions.

The set of constraints for a given statement or expression is denoted as $\mathcal{C}^m[E^\ell]$ of simple constraints, using the types of constraints described below. Here E is the expression or PHP statement, ℓ the label of this expression or statement, and m the label of the enclosing statement in which E appears (or the label of E itself if E is a statement). For example, $\mathcal{C}^\ell[S^\ell]$ is the set of constraints for statement S which has label ℓ. Each constraint is expressed using the labels (e.g. ℓ) of each statement or expression. Because expressions are not independent entities in a PHP program (they are always part of a statement), we refer to them with $\mathcal{C}^\star[e^\ell]$ instead of $\mathcal{C}^\ell[e^\ell]$, when the label of enclosing statement is unknown.

There are three types of constraints in our type inference system:

$\{\tau_1, ..., \tau_n\} \subseteq type(\ell)$ The constraint $\{\tau_1...\tau_n\} \subseteq type(\ell)$ states that given an expression e with label ℓ, it yields the exact type set $\{\tau_1...\tau_n\}$ for e. Most often, this type set only consists of a singleton type, in which case it is easier to write $\ell := \tau$.

$\ell \equiv \tau$ The constraint $\ell \equiv \tau$ does not introduce new types for ℓ, but it makes the assertion that a type τ is *expected* for ℓ. This does not mean that ℓ will actually be of this type, and so the assertion may *fail*. We use these constraints to detect (possible hazardous) coercions. If more than one expected type is possible for ℓ, we write $\ell \subseteq_\equiv \{\tau_1...\tau_n\}$.

[3] We assume here that our explicit order of global and static variable declaration, mentioned in Section 6.2, is applied.

$\mathcal{C}^\star[\texttt{null}^\ell] = \{\ell := \bot\}$
$\mathcal{C}^\star[\texttt{int}^\ell] = \{\ell := \mathit{int}\}$ (e.g. $\mathcal{C}^\star[\texttt{3}^\ell] = \{\ell := \mathit{int}\}$)
$\mathcal{C}^\star[\texttt{float}^\ell] = \{\ell := \mathit{float}\}$ (e.g. $\mathcal{C}^\star[\texttt{6.5}^\ell] = \{\ell := \mathit{float}\}$)
$\mathcal{C}^\star[\texttt{true}^\ell] = \{\ell := \mathit{bool}\}$, $\mathcal{C}^\star[\texttt{false}^\ell] = \{\ell := \mathit{bool}\}$
$\mathcal{C}^\star[\texttt{string}^\ell] = \{\ell := \mathit{string}\}$ (e.g. $\mathcal{C}^\star[\texttt{"test"}^\ell] = \{\ell := \mathit{string}\}$)

Figure 7.4: Constraints for constant values

$\ell_1 \Leftarrow \ell_2$ The constraint $\ell_1 \Leftarrow \ell_2$ implies a data-flow from ℓ_2 to ℓ_1. This means that all constraints that are valid for ℓ_2, are also valid for ℓ_1. However, this applies to both of the previously described constraint types, so a distinction has to be made: if all *expected* types of ℓ_2 are also expected for ℓ_1, we write $\ell_1 \Leftarrow_\equiv \ell_2$. In the other case we can just write $\ell_1 \Leftarrow \ell_2$. However, if ℓ_2 may have multiple types, we just write $\mathit{type}(\ell_2) \subseteq \mathit{type}(\ell_1)$ instead.

In the remainder of this section, the constraints which are generated for each language construct will be discussed. Although the sub-elements of a given language construct (e.g. the operands of a binary expression) may generate more constraints themselves, we only give the new constraints generated specifically by this language construct (the *delta*). An example:

Consider the statement $\texttt{\$a}^{\ell_1} = (\texttt{\$b * 3})^{\ell_2}$. The constraints for this statement are $\mathcal{C}^\ell[(\texttt{\$a}^{\ell_1} = (\texttt{\$b} * 3)^{\ell_2})^\ell] = \{\ell_1 \Leftarrow \ell_2\} \cup \mathcal{C}^\ell[\texttt{\$a}^{\ell_1}] \cup \mathcal{C}^\ell[(\texttt{\$b}^{\ell_3} * 3^{\ell_4})^{\ell_2}]$. Only the constraint $\ell_1 \Leftarrow \ell_2$ is explicitly generated by the statement. The set $\mathcal{C}^\ell[(\texttt{\$b}^{\ell_3} * 3^{\ell_4})^{\ell_2}]$ generates the constraint $\ell_2 := \mathit{num}$, and also contains the constraint sets for ℓ_3 and ℓ_4. A more elaborate example of constraint generation – of a complete program – can be found in Section 7.4.

Constants and literals

Constraints for constants are all of the form $\ell := \tau$, where each kind of constant (as given by Figure 6.1) has its own distinct type. Figure 7.4 lists an overview of all constraints for constants. Constants only generate one constraint each.

Expressions and operators

Depending on the kind of expression, and its operator, expressions typically generate two or three additional constraints. Binary operators (and most unary operators as well) involve two kinds of constraints. One $\ell \equiv \tau$ constraint is generated for each operand, and a $\ell_1 \Leftarrow \ell_2$ constraint is generated for its result type. Figure 7.5 lists the constraints for expressions that do not involve an operator and all expressions with unary operators. Figure 7.6 lists the constraints for expressions involving multiple operands.

One interesting observation that can be made from Figure 7.6 is that we assign *num* to numeric operations, like e.g. `3 + 4.5`, instead of their exact types, i.e. *int* or *float*. We have chosen for this approach because we do not attach much value to the fact if a variable is either a *int* or a *float* specifically, but only that they are *numeric*.

Statements

Now that we have defined the constraints for expressions, defining the constraints for statements becomes straightforward. The most common ones are assignments, choice statements and loops. In Figure 7.7 the constraints for assignments, printing statements like `echo`, and various others that do not use an elaborate form of control structure are listed. Compound statements generate no constraints themselves, but the inner statement(s) can.

Include statements are handled differently. Because includes are handled *by the parser*, it is not required to generate any constraints for these, because their contents have already been inserted into the abstract syntax tree directly.

$\mathcal{C}^\star[(v^{\ell_1}[e^{\ell_2}])^\ell] = \{\ell_1 \equiv array[any], \ell \equiv fromarray(\ell_1), \{int, string\} \subseteq_\equiv type(\ell_2)\}$
$\mathcal{C}^\star[(v^{\ell_1} \texttt{ = } e_2^{\ell_2})^\ell] = \{\ell_1 \Leftarrow \ell_2, \ell \Leftarrow \ell_2\}$

$\mathcal{C}^\star[(\texttt{ (}e^\ell\texttt{) })] = \emptyset$

$\mathcal{C}^\star[(e^{\ell_1}\texttt{--})^\ell] = \{\ell_1 \equiv int, \ell := int\}$, $\mathcal{C}^\star[(e^{\ell_1}\texttt{++})^\ell] = \{\ell_1 \equiv int, \ell := int\}$
$\mathcal{C}^\star[(\texttt{--}e^{\ell_1})^\ell)] = \{\ell_1 \equiv int, \ell := int\}$, $\mathcal{C}^\star[(\texttt{++}e^{\ell_1})^\ell)] = \{\ell_1 \equiv int, \ell := int\}$

$\mathcal{C}^\star[(\texttt{!}e^{\ell_1})^\ell)] = \{\ell_1 \equiv bool, \ell := bool\}$
$\mathcal{C}^\star[(-e^{\ell_1})^\ell)] = \{\ell_1 \equiv num, \ell := num\}$
$\mathcal{C}^\star[((\tau)e^{\ell_1})^\ell)] = \{\ell := \tau\}$ where $\tau \in base\_type$
$\mathcal{C}^\star[(\texttt{count}(e^{\ell_1})^\ell] = \{\ell := int, \ell_1 \equiv array[any]\}$
$\mathcal{C}^\star[(\texttt{array}(e_1^{\ell_1}, ..., e_n^{\ell_n}))^\ell] = \{array[\tau] \subseteq type(\ell) : \ \tau \in type(\ell_i) \mid i \in \{1...n\}\}$

Figure 7.5: Constraints for expressions without operators or using unary operators

$\mathcal{C}^\star[(e_1^{\ell_1} \oplus e_2^{\ell_1})^\ell] = \{\ell_1 \equiv num, \ell_2 \equiv num, \ell := num\}$ where $\oplus \in \{\texttt{+}, \texttt{ -}, \texttt{ *}, \texttt{ /}, \texttt{ \%}\}$
$\mathcal{C}^\star[(e_1^{\ell_1} \texttt{ . } e_2^{\ell_2})^\ell] = \{\ell_1 \equiv string, \ell_2 \equiv string, \ell := string\}$
$\mathcal{C}^\star[(e_1^{\ell_1} \texttt{ == } e_2^{\ell_2})^\ell] = \{\ell_1 \Leftarrow_\equiv \ell_2, \ell := bool\}$
$\mathcal{C}^\star[(e_1^{\ell_1} \texttt{ != } e_2^{\ell_2})^\ell] = \{\ell_1 \Leftarrow_\equiv \ell_2, \ell := bool\}$
$\mathcal{C}^\star[(e_1^{\ell_1} \oplus e_2^{\ell_2})^\ell] = \{\ell_1 \equiv num, \ell_2 \equiv num, \ell := bool\}$ where $\oplus \in \{\texttt{<}, \texttt{ >}, \texttt{ <=}, \texttt{ >=}\}$
$\mathcal{C}^\star[(e_1^{\ell_1} \oplus e_2^{\ell_2})^\ell = \{\ell_1 \equiv bool, \ell_2 \equiv bool, \ell := bool\}$ where $\oplus \in \{\texttt{\&\&}, \texttt{ ||}\}$

$\mathcal{C}^\star[(e_1^{\ell_1} \texttt{ ? } e_2^{\ell_2} \texttt{ : } e_3^{\ell_3})^\ell] = \{\ell_1 \equiv bool, \ell_2 \subseteq type(\ell), \ell_3 \subseteq type(\ell)\}$

Figure 7.6: Constraints for expressions involving multiple operands

$\mathcal{C}^\ell[(\texttt{echo } e_1^{\ell_1}...e_n^{\ell_n})^\ell] = \{\ell_i \equiv string \mid i \in \{1...n\}\}$

$\mathcal{C}^\ell[(v^{\ell_1} = e^{\ell_2})^\ell] = \{\ell_1 \Leftarrow \ell_2\}$,
$\mathcal{C}^\ell[(v^{\ell_1}[e_1^{\ell_2}] = e^{\ell_3})^\ell] = \{\ell_1 \Leftarrow array(\ell_3), \ell_2 \subseteq_\equiv \{int, string\}\}$,
$\mathcal{C}^\ell[(v_1^{\ell_1} = v_2^{\ell_2}[e^{\ell_3}])^\ell] = \{\ell_1 \Leftarrow fromarray(\ell_2), \ell_2 \equiv array[any], \ell_3 \subseteq_\equiv \{int, string\}\}$,
$\mathcal{C}^\ell[(v_1^{\ell_1}[e_1^{\ell_2}] = v_2^{\ell_3}[e_2^{\ell_4}])^\ell] = \{\ell_1 \Leftarrow \ell_3, \ell_2 \subseteq_\equiv \{int, string\}, \ell_4 \subseteq_\equiv \{int, string\}\}$, where

- $array(\ell)$ lifts the type of ℓ to an array of that type (i.e. τ to $array[\tau]$). The definition of $array(\ell)$ is: $array(\ell) = array[type(\ell)]$.
- $fromarray(\ell)$ gives the base type from the array type of ℓ (i.e. the reverse of $array(\ell)$).
- Multi-dimensional arrays are not discussed here. However, these are handled exactly as displayed above, where the expression between brackets (e.g. $[e^{\ell_3}]$) represents the last index. This means that an expression such as $v_1^{\ell_1}$ could represent both a variable, as well as an array reference if it is referred to with an array index expression.

$\mathcal{C}^\ell[(v^{\ell_1} \oplus e^{\ell_2})^\ell] = \{\ell_1 := num, \ell_2 \equiv num\}$ where $\oplus \in \{\texttt{+=}, \texttt{ -=}, \texttt{ *=}, \texttt{ /=}, \texttt{ \%=}\}$
$\mathcal{C}^\ell[(v^{\ell_1} \oplus e^{\ell_2})^\ell] = \{\ell_1 := string, \ell_2 \equiv string\}$ where $\oplus \in \{\texttt{.=}, \texttt{ ~=}\}$
$\mathcal{C}^\ell[(v^{\ell_1} \oplus e^{\ell_2})^\ell] = \{\ell_1 := bool, \ell_2 \equiv bool\}$ where $\oplus \in \{\texttt{\&=}, \texttt{ |=}, \texttt{ ^=}\}$, where

- Arrays in these constraints are handled similarly as above.

$\mathcal{C}^\ell[(e^{\ell_1}\texttt{--})^\ell] = \{\ell_1 \equiv int\}$, $\mathcal{C}^\ell[(e^{\ell_1}\texttt{++})^\ell] = \{\ell_1 \equiv int\}$
$\mathcal{C}^\ell[(\texttt{--}e^{\ell_1})^\ell)] = \{\ell_1 \equiv int\}$, $\mathcal{C}^\ell[(\texttt{++}e^{\ell_1})^\ell)] = \{\ell_1 \equiv int\}$

Figure 7.7: Constraints for basic statements

$\mathcal{C}^{\ell}[(\texttt{if (}\ e^{\ell_1}\ \texttt{)}\ S^{\ell_2})^{\ell}] = \{\ell_1 \equiv \mathit{bool}\}$
$\mathcal{C}^{\ell}[(\texttt{if (}\ e^{\ell_1}\ \texttt{)}\ S_1^{\ell_2}\ \texttt{else}\ S_2^{\ell_3})^{\ell}] = \{\ell_1 \equiv \mathit{bool}\}$

$\mathcal{C}^{\ell}[(S_1^{\ell_1}\ \texttt{or}\ S_2^{\ell_2})^{\ell}] = \emptyset$

$\mathcal{C}^{\ell}[(\texttt{ switch (}\ e^{\ell_e}\ \texttt{) \{case}\ v_1^{\ell_1}, ..., \texttt{case}\ v_n^{\ell_n}\texttt{\})}^{\ell}] = \{\ell_e \Leftarrow \ell_i \mid i \in \{1...n\}\}$

Figure 7.8: Constraints for control structures

$\mathcal{C}^{\ell}[(\texttt{while (}\ e^{\ell_1}\ \texttt{)}\ S^{\ell_2}\ \texttt{)}^{\ell}] = \{\ell_1 \equiv \mathit{bool}\}$
$\mathcal{C}^{\ell}[(\texttt{do}\ S^{\ell_1}\ \texttt{while (}\ e^{\ell_2}\ \texttt{))}^{\ell}] = \{\ell_2 \equiv \mathit{bool}\}$

$\mathcal{C}^{\ell}[(\texttt{for (}\ \mathit{decls}^{\ell_d}\ \texttt{;}\ e_1^{\ell_1}\ ...\ e_n^{\ell_n}\ \texttt{;}\ \mathit{incs}^{\ell_i}\ \texttt{)}\ S^{\ell_s}\ \texttt{)}^{\ell}] = \{\ell_k \equiv \mathit{bool} \mid k \in \{1...n\}\}$

$\mathcal{C}^{\ell}[(\texttt{foreach (}e^{\ell_1}\ \texttt{as}\ e^{\ell_2})S)^{\ell}] = \{\ell_1 \equiv \mathit{array}[\mathit{any}], \ell_2 \Leftarrow \mathit{fromarray}(\ell_1)\}$
$\mathcal{C}^{\ell}[(\texttt{foreach (}e^{\ell_1}\ \texttt{as}\ e^{\ell_2} \texttt{=>}\ e^{\ell_3})S)^{\ell}] = \{\ell_1 \equiv \mathit{array}[\mathit{any}], \ell_2 \subseteq_{\equiv} \{\mathit{int}, \mathit{string}\}, \ell_3 \Leftarrow \mathit{fromarray}(\ell_1)\}$

Figure 7.9: Constraints for loops

Choice structures

Choice structures add a new dimension to our type inference system: branching[4]. For type inference, this means we have to compute the *union* of the types for each variable at the end of this statement; each branch is required to compute the results for the entire structure.

In Figure 7.8 the constraints for `if ... else`, `... or ...`, and the `switch` constructs are given.

Loops

Even though the flow of loop statements is more advanced than that of other statements, they actually do not generate many constraints. Figure 7.9 gives the constraints for all of the four loop structures.

Functions and function calls

Functions can be declared at any location, but our restrictions in Section 6.2 ensure that they only appear at top-level scope. This ensures that whenever a function is declared, its definition is unique, because redeclaration of functions is prohibited in PHP [24].

In Section 6.1.6 we explained that there are two scopes in PHP, a global scope, and a function scope. When inside of a function, there is no access to any of the variables outside of this function, unless these variables are declared as *global*. Static variables are stored after each function call, so these can also bring extra information. Global variable declarations do not require any constraints at all, because these are directly handled by the constraint solver, which we discuss in Section 7.4. The constraints for static variables are generated directly at their declaration, before the body of the function starts, if no value for a static variable is not yet already stored. Constraints for static variables are generated like if these are assignments.

In Figure 7.10, the constraints for function calls and returns are displayed. A set of constraints for a function call is a set $\mathcal{C}^{\ell_1}[f(e_\star^{\ell_3})^{\ell_1,\ell_2}]$, where ℓ_1 is the label for the function call statement, f is the name of the function to be called, and $e_\star$ its arguments. The labels ℓ_1 and ℓ_2 are the call and return labels for this function call. Looking at the constraints set for the results of a function, we get a set $\mathcal{C}^{\ell_2}[f(p_\star^{\ell_3})^{\ell_1,\ell_2}]$, where $p_\star$ are the parameters of this function that correspond to the arguments $e_\star$ of the function call. These constraints model the effect of returning to the location of the function call, hence the return label is used as the reference point.

[4] Actually, the conditional *expression*, namely e_1 ? e_2 : e_3 is also a choice structure with branches. But that is branching on the *expression level*, where we speak of the type set of a single expression or variable. This is branching on the *statement level*, where branching may involve many expressions, variables and other statements.

$\mathcal{C}^{\ell_{n+1}}[f(e_1^{\ell_1}, ..., e_n^{\ell_n})^{\ell_{n+1},\ell_{n+2}}] = \{x_1 \Leftarrow \ell_1, ..., x_n \Leftarrow \ell_n\} \cup \{x_{n+1} \Leftarrow p_1, ..., x_{n+m} \Leftarrow p_m\}$ where
- $x_1, ...x_n$ are the labels for the parameters of function f,
- $x_{n+1}, ...x_{n+m}$ are the labels for the default parameters for function f, and
- $p_1, ..., p_m$ are the labels for the default parameters' expressions of function f

$\mathcal{C}^{\ell_2}[$ (`return` $e^{\ell_1})^{\ell_2}] = \{\ell_1 \subseteq type(result_f)\}$ where
- f is the function in which this `return` statement occurs,
- $result_f$ is the result label of function f

$\mathcal{C}^{\ell_{n+2}}[f(e_1^{\ell_1}, ..., e_n^{\ell_n})^{\ell_{n+1},\ell_{n+2}}] = \{\ell_{n+2} \Leftarrow result_f\}$ where
- $result_f$ is the result label of function f,

Figure 7.10: Constraints for function calls, `return` statement, and function result

The actual function type is not produced by the constraints themselves. It is constructed from the *results* calculated from these constraints by the solving algorithm, discussed in Section 7.4.

The constraints for function calls, the return statement, and function result are given in Figure 7.10. If the called function however is a *library function* – meaning it has been defined in a public library of which the source code is not available –, or is undefined, these constraints cannot be generated, because nothing is known about the internals of the function – i.e. the parameters, default arguments, global and static variables. For functions such as these we only infer the argument types, and the return type (if possible).

7.4 Constraint solving

In Section 7.3 we presented the sets of constraints that are derived for each of the language constructs. In this section, we present an algorithm to solve these constraints and actually derive type sets for each of the variables and functions in the PHP program. We illustrate it using a few examples. The definitions and theories about contexts, transfer functions and algorithms for Monotone Frameworks is discussed in [39]. We use the worklist algorithm described by them, which has been altered to work with PHP.

7.4.1 Definitions

Before we can describe our algorithm, we need to establish some definitions first. Our algorithm assigns type sets to identifiers, which are recorded in a type environment for each program point. We define the type environment $TEnv$ as follows:

$TEnv : Id \rightarrow type\_set$, where
$Id \in Var \cup FunctionId$

Here Var is the set of all possible variables, and $FunctionId$ the set of all possible function identifiers. Var and $FunctionId$ are disjoint sets; an identifier can be in only one of of these two sets.

Our algorithm iterates on the program flow, using the constraint set definitions that we defined in Section 7.3. This program flow is specified only on *statement level*: the effects of expressions (e.g. assignment expressions) are accumulated in their evaluation order, which is from left to right, where inner expressions are evaluated first (i.e. a post-order traversal). The entry point of the program F_0 is given to the algorithm, as well as the rest of the program flow.

Some nodes (i.e. ending nodes of control structures) do not cause any changes in the type sets, yet are physically present in our ASTs[5]. We call these *skip nodes*. Empty compound statements (e.g. for `if ... else` without an `else` clause) are considered to be skip nodes as well. Skip nodes have no influence on the analysis, i.e., their transfer function is the *identity* function.

For the iteration we use a worklist, which is implemented as a stack with the operations *pop*() to take the top element of the stack, and the *push*(*elem*) method to place a new element *elem* on top of it. Any other data

[5]They are however *not* physically present in the PHP code. We added these nodes for simplicity in our AST.

INPUT: The set of constraints $\mathcal{C}^\star[E_\star]$, the set of program points $P_\star$, the unification (widening) operator ∇, the entry point F_0

OUTPUT: $analysis : P \rightarrow TEnv \times TEnv \times TEnv$

METHOD:

Step 1: Initialization

F := *flow graph from* $P_\star$;
$W.push(F_0)$;
`foreach` p `in` $P_\star$ `do` $analysis(p) := (\emptyset, \emptyset, \emptyset)$;

Step 2: Iteration

`while` $W \neq nil$ `do`
$edge := W.pop()$; $node := edge.to$;
$(entry, exit, \_) := analysis(node)$;
$constraints :=$ set of constraints $\mathcal{C}^{node}$, derived from $\mathcal{C}^\star[E_\star]$, without expected types;
$new\_set := transfer\_types(entry, constraints, \nabla)$;
$exit' := exit \sqcup new\_set$;
`if` $exit' \neq exit$ `then`
$exit := exit'$;
`foreach` $node' \in P_\star : (node, node') \in F$ `do`
$W.push(node, node')$;
$(entry', \_, \_) := analysis(node')$;
$entry' := entry' \sqcup exit'$;

Step 3: Calculate the expected types

`foreach` $node \in P_\star$:`if` $analysis(node) \neq \bot$ `then`
$(\_, \_, expected) := analysis(node)$;
$constraints :=$ set of constraints $\mathcal{C}^{node}$, derived from $\mathcal{C}^\star[E_\star]$, with only expected types;
$expected := calculate\_expected(constraints, \nabla)$;

Step 4: Return the solution

`return` $analysis$;

Figure 7.11: The algorithm without function calls and contexts

structure can be used for the worklist, but the stack has some properties that prove to be especially useful (as we will see in Section 7.4.3). The elements of this stack are edges of the program flow. Each edge has two pointers, to the previous and current node.

During iteration, the transfer function uses a widening operator ∇ (e.g. the one that we defined in Section 7.2), to unify the current type sets with the newly inferred ones. In the algorithm, these newly inferred sets of variables (with their types) are then unified using the $\sqcup$ operator, with the variable sets that were found during previous iterations. The $\sqcup$ operator takes two type environments of type $TEnv$ and returns a new environment, which is the unification of these environments. The ∇ operator is used to merge each type set for each identifier specifically.

The result of the algorithm is a function *analysis* which gives for a given program point, two sets containing the types for each identifier before and after executing the statement of this program point (i.e. its *entry* and *exit* sets), and a third set with its expected types. The function *analysis* is defined as follows:

$analysis : P \rightarrow TEnv \times TEnv \times TEnv$

Here P is a program label from the program flow (statement, loop or control construct), and $TEnv \times TEnv \times TEnv$ the triple containing the entry, exit and expected type sets.

7.4.2 The basic algorithm

In Figure 7.11 we present the worklist algorithm for programs without function calls, which consists of four steps.

```
function transfer_types(entry, constraints, ∇) is
    types := entry;
    foreach cc ∈ resolve_dependencies(constraints) do
        case cc of
            ℓ := τ : if ℓ ∈ Id then types(ℓ) := τ;
            {τ1...τn} ⊆ type(ℓ) : if ℓ ∈ Id then types(ℓ) := types(ℓ)∇{τ1...τn};
    return types;

function resolve_dependencies(constraints) is
    new_constraints := constraints;
    while new_constraints has constraints with a dependency between a label ℓ1 and ℓ2 do
        cons := ℓ1 ⇐ ℓ2 ∈ new_constraints;
        foreach ℓ2 := τ ∈ constraints do new_constraints.add(ℓ1 := τ);
        foreach ℓ2 := ℓ3 ∈ constraints do new_constraints.add(ℓ1 := ℓ3);
        foreach {τ1...τn} ⊆ ℓ2 ∈ constraints do new_constraints.add({τ1...τn} ⊆ ℓ1);
    return new_constraints;


function calculate_expected(constraints, ∇) is
    types := ∅;
    foreach cc ∈ resolve_dependencies_for_expected_types(constraints) do
        case cc of
            ℓ ≡ τ : if ℓ ∈ Id then types(ℓ) := τ;
            {τ1...τn} ⊆≡ type(ℓ) : if ℓ ∈ Var then types(ℓ) := types(ℓ) ∪ {τ1...τn};
    return types;

function resolve_dependencies_for_expected_types(constraints) is
    new_constraints := constraints;
    while new_constraints has constraints with dependencies between a label ℓ1 and ℓ2 do
        cons := ℓ1 ⇐≡ ℓ2 ∈ new_constraints;
        foreach ℓ2 ≡ τ ∈ constraints do new_constraints.add(ℓ1 ≡ τ);
        foreach ℓ2 ≡ ℓ3 ∈ constraints do new_constraints.add(ℓ1 ≡ ℓ3);
        foreach {τ1...τn} ⊆≡ ℓ2 ∈ constraints do new_constraints.add({τ1...τn} ⊆≡ ℓ1);
    return new_constraints;
```

Figure 7.12: Definitions of functions for calculating transfer and expected types

In the first step, F is initialized with the program's flow graph. The worklist W is initialized with the initial edge F_0, which is an edge from the start label of the program to the first statement. Next, we initialize each of the entries for the result function *analysis* with a triple of empty sets.

In the second step, iteration is performed for as long as W still contains edges that need to be analysed. With the $transfer$ function, a new set of types for each identifier is calculated, using the *entry* set and the constraints for the current statement's label, which is *node*. We then unify this new set of types using $\sqcup$ with *exit* to obtain $exit'$, which is then compared to *exit*. If $exit'$ differs from *exit*, then this iteration had a new effect on the type sets of *node*, and as a result all nodes that succeed *node* in the program flow will have to be (re)analysed, and are added to the worklist W. If *node* is a skip node, then the transfer function is the identity function, and will only propagate information from the previous node.

In the third step, the expected types are calculated. Because the *expected types* never change when the results for an edge in the program flow are recalculated again, it is desirable to leave these out of the fix-point iteration, and calculate these afterward, for every program point that was reached during the fix-point calculation (as these contain the program points that may actually be executed). Finally in the fourth and last step the function *analysis* giving the entry and exit sets for every program point $p \in P_\star$ is returned, as result.

In Figure 7.12 we define $transfer\_types$, which is used in our algorithm in Figure 7.11. This function simply iterates through all the constraints, modifying the type set *entry* to satisfy all the constraints. All constraints are *flattened* first – i.e. all dependencies between constraints are resolved using the *resolve_dependencies* function. The function *calculate_expected* calculates the expected types, to satisfy the constraints that impose these types.

To illustrate our algorithm, consider the following program:

```
<?ℓ1
  $aℓ2 = 0ℓ3;ℓ4
  $bℓ5 = 0ℓ6;ℓ7
  whileℓ8,28 ($aℓ9 < 10ℓ10)ℓ11{
    $aℓ12++;ℓ13
    ifℓ14,27 ( ($bℓ15 % 3ℓ16)ℓ17 == 0ℓ18)ℓ19
      $bℓ20 += ($aℓ21 * 2ℓ22)ℓ23;ℓ24
    else
      $bℓ25--;ℓ26
  }
  $cℓ29 = ( ($bℓ30 < 10ℓ31)ℓ32 ? "test"ℓ33 : trueℓ34)ℓ35;ℓ36
  echo $cℓ37;ℓ38
?>ℓ39
```

The constraints for this example are calculated using the definitions from Section 7.3:
$\mathcal{C}^{\ell_4} = \{\ell_3 := int, \ell_2 \Leftarrow \ell_3\}$,
$\mathcal{C}^{\ell_7} = \{\ell_6 := int, \ell_5 \Leftarrow \ell_6\}$,
$\mathcal{C}^{\ell_8} = \{\ell_{11} \equiv bool, \ell_9 \equiv num, \ell_{10} \equiv num, \ell_{10} := int\}$,
$\mathcal{C}^{\ell_{13}} = \{\ell_{12} \equiv int\}$,
$\mathcal{C}^{\ell_{14}} = \{\ell_{15} \equiv num, \ell_{16} \equiv num, \ell_{16} := int, \ell_{17} := int, \ell_{17} \Leftarrow \ell_{18}, \ell_{18} := int, \ell_{19} \equiv bool, \ell_{19} := bool\}$,
$\mathcal{C}^{\ell_{24}} = \{\ell_{21} \equiv num, \ell_{22} \equiv num, \ell_{22} := int, \ell_{20} := num, \ell_{23} \equiv num\}$,
$\mathcal{C}^{\ell_{26}} = \{\ell_{25} \equiv int\}$,
$\mathcal{C}^{\ell_{36}} = \{\ell_{30} \equiv num, \ell_{31} \equiv num, \ell_{30} := int, \ell_{32} \equiv bool, \ell_{32} := bool, \ell_{33} \equiv string, \ell_{34} \equiv bool, \ell_{33} \subseteq \ell_{35}, \ell_{34} \subseteq \ell_{35}, \ell_{29} \Leftarrow \ell_{35}\}$,
$\mathcal{C}^{\ell_{38}} = \{\ell_{37} \equiv string\}$.

When running the algorithm, these constraints will have to be satisfied. For example, when the iteration reaches the statement with the assignment to `$c`, (ℓ_{36}), the situation at the start of the iteration step could be:

$edge = (\ell_{28}, \ell_{36})$,
$entry = \{\texttt{\$a} \mapsto \{int\}, \texttt{\$b} \mapsto num\}$,
$exit = \emptyset$,
$constraints = \mathcal{C}^{\ell_{36}}$.

The algorithm then applies $transfer\_types$ to $entry$, using the constraints to get new_set, and merges it with $exit$ to obtain $exit'$. The value of $exit'$ is:

$exit' = \{\$a \mapsto \{int\}, \$b \mapsto num, \$c \mapsto \{string, bool\}\}$;

This set is different from $exit$, so iteration will have to proceed on the successor nodes, in this case only ℓ_{38}. The edge of ℓ_{36} to ℓ_{38} will be added to the stack W, and the $exit'$ of statement ℓ_{36} will be merged into the entry set for ℓ_{38}.

After fix-point iteration, the expected types are calculated. This will result in the following value for this specific node (i.e. ℓ_{36}):

$expected = \{\$b \mapsto num\}$

This expected type num matches the type found in the fix-point iteration perfectly, so no problem is found for this variable. However, $c has two types, which will result in a warning (as discussed in Section 7.5).

Now, the next step is to involve function calls into the algorithm. Function calls add a lot of complications, for example context information, and proper handling of recursion. We discuss this in the next section.

7.4.3 Adding functions to the algorithm

Functions add some extra complexity to type analysis, due to the fact that they imply a need for inter-procedural analysis, and information about the various function call locations. The principles of inter-procedural analysis are explained in [39]. In this section we apply these principles to our type inference system for PHP.

Functions can be called from both *statement level* and *expression level*. Flow to function calls on expression level should be processed before the statement in which this expression occurs, because these calls can influence the statement as a whole.

The call location of a function call is important, because different function calls may use different argument types, and this may result in a different result type for different calls of the same function. The call location of a function call is administered in something we call a context, and the usual representation of these is a call string, which is simply a call stack or history, containing the locations from where the function was called. However, in recursive functions, this call string can become arbitrarily large, and unbounded call strings are not desirable. A form of limiting the length of a call string is recording the information of the most recent k calls, where k is a positive number. The larger k is chosen, the more precise the results of the analysis will be. Another form is only recording the names of each function that has been called. This does not impose a physical limit on the call string's length, but it may preserve more information than the k-bounded call strings. However, for simplicity we will consider the k-bounded call strings in our algorithm.

Because program nodes now involve both contexts and program nodes, the definition of $analysis$ needs to be enhanced. The definition of $analysis$ is now:

$analysis : P \to C \to TEnv \times TEnv \times TEnv$

Here C is the context information, represented by, for example, a call string, P a program label of our script (statement, loop or control construct), $TEnv \times TEnv \times TEnv$ a triple of type environments representing the entry, exit and expected type sets for P in context C.

Also, our definition of the set of identifiers Id needs to be enhanced. Because function calls have arguments, and the types of these need to be passed into the called function, argument identifiers have to be introduced. We call the set of argument identifiers $CallArgs$, and then the new definition of Id will then be:

$Id \in Var \cup FunctionId \cup CallArgs$

Identifiers from $CallArgs$ are identifiers that are not physically present in the program code, but are artificial ones, used to identify and distinguish the call parameters for a specific call. These are used to pass the call argument type information to the function parameters, and to define the complete type of the function, after the function call has been completely analysed. Afterward, they are no longer present in the analysis results.

In Figure 7.13 we present our algorithm from Section 7.4.2, augmented to handle function calls and contexts. The algorithm still consists of four steps.

INPUT: The set of constraints $\mathcal{C}^\star[E_\star]$, the set of program points $P_\star$, the unification (widening) operator ∇, the entry point F_0

OUTPUT: $analysis : P \to C \to TEnv \times TEnv \times TEnv$

METHOD: Step 1: Initialization

$F :=$*flow graph from* $P_\star$;
$W.push(F_0)$;
`foreach` p `in` $P_\star \wedge p$ is not inside a function `do` $analysis(p) := (\emptyset, \emptyset, \emptyset)$;

Step 2: Iteration

`while` $W \neq nil$ `do`
$edge := W.pop()$; $node := edge.to$; $prev := edge.from$;
$context := edge.to\_context$; $prev\_context := edge.from\_context$;
$(entry, exit, \_) := analysis(node)(context)$;
$constraints :=$ set of constraints $\mathcal{C}^{node}$, derived from $\mathcal{C}^\star[E_\star]$, without expected types;
$new\_set := transfer\_types(entry, constraints, \nabla)$;
$exit' := exit \sqcup new\_set$;

`if` *node* is a call entry node `then`
$exit := exit'$; $function := node.function$;
`if` $function \neq \bot$ `then`
$new\_context := mutate\_context(context, node)$;
$W.push((new\_context, function.exit), (context, node.return)$;
$W.push((context, node), (new\_context, function.init)$;
`else` $W.push((context, node), (context, node.return)$;
$(entry', \_, \_) := analysis(node.return)(context)$;
$entry' := entry' \sqcup exit'$;
`if` *node* is a function exit node `then`
$exit := exit'$;
`else if` $exit' \neq exit$ `then`
$exit := exit'$;
`foreach` $node' \in P_\star : (node, node') \in F$ `do`
$W.push((context, node), (context, node'))$;
$(entry', \_, \_) := analysis(node')(context)$;
$entry' := entry' \sqcup exit'$;

Step 3: Calculate the expected types

`foreach` $node \in P_\star$ `:if` $analysis(node) \neq \bot$ `then`
$constraints :=$ set of constraints $\mathcal{C}^{\ell_{node}}$, derived from $\mathcal{C}^\star[E_\star]$ with only expected types;
`foreach` $(\_, \_, expected) \in analysis(node)$ `do` $expected := calculate\_expected(constraints, \nabla)$;

Step 4: Return the solution

`return` *analysis*;

Figure 7.13: A worklist algorithm with function calls and contexts

```
function transfer_types(edge, entry, constraints, ∇) is
    types := entry;
    node := edge.to;
    foreach cc ∈ resolve_dependencies(constraints) do
        case cc of
            ℓ := τ : if ℓ ∈ Id then types(ℓ) := τ;
            {τ1...τn} ⊆ type(ℓ) : if ℓ ∈ Id then types(ℓ) := types(ℓ)∇{τ1...τn};
    if node is a function init node then
        function := node.function; i := 0;
        (_, prev_exit, _) := analysis(edge.from)(edge.from_context);
        argument_types := ∅;
        foreach v ∈ entry ∧ v ∈ CallArgs do argument_types(v) := entry(v);
        for v ∈ function.params do
            if i ≤ |argument_types| then types(v) := argument_types(i);
            i := i + 1;
    else if node is a call return node then
        function := node.function;
        argument_types := ∅;
        foreach v ∈ entry ∧ v ∈ CallArgs do argument_types(v) := entry(v);
        types := ∅;
        foreach v ∈ entry ∧ v ∉ CallArgs do types(v) := entry(v);
        if function ≠⊥ then
            result_type := function_list(function, argument_types);
        else
            result_type := types(result_function);
            if result_type := ∅ then result_type := {void};
        types(function) := types(function) ∪ {argument_types → result_type};
return types;
```

Figure 7.14: The transfer function for our worklist algorithm with function calls

The first step in the new algorithm is exactly the same as for the algorithm without function calls, and *analysis* is now initialized for all nodes that do not appear inside a function (instead of all nodes). In the second step iteration is performed for as long as W still contains edges that need to be analysed. A clear distinction is now made for when a entry node of a function call and the exit node of a function is being analysed.

Because we include the context information within the worklist W, we only have to update the current context when a function call is encountered. This function, *mutate_context*, simply takes the current context and the call entry node, and adds the call entry node's information to the call string, returning a new context. This new context may be exactly the same as the current context though, which will lead to type sets of previous function calls becoming unified with those of new ones. We also make use of the LIFO (last in, first out) property of the stack here: we add the edge of the function exit to the call return first, and then the edges of the body of the function. This ensures us that, if we finish analysing a function body, we always have an edge leading to the call return for this function call, with the correct context information.

We only analyse locally defined functions; library functions and undefined functions will be skipped, and have their call entry node be directly connected to the call exit node. The result types for library function are fetched from a predefined list, for undefined functions we assume *any* as its result type (because we do not know the actual type).

In the third step the expected types are calculated, and finally in the last step the result function *analysis* is returned.

The iteration step uses a transfer function, presented in Figure 7.14, which carries a lot more responsibilities than in the algorithm without function calls. Different actions have to be taken if the current node is a function entry node or a call end node. In the other cases, the usual behaviour is performed.

For a function entry node, the parameter types are read from the call argument variables. If a parameter is used as a *default parameter*, an argument variable for this parameter does not exist, and its type will be provided by the regular constraint solving.

For a call end node, the call arguments are taken from the exit set of the call entry node. The result type of the function is copied from the function exit node's exit set, if available. If the function was undefined, the function result is possibly a predefined (library) function, and retrieved from a function list with all types of these functions. If no return type can be found at all (if the function is defined), then it apparently is a procedure, and then the *void* type is the result type. Finally, the function type $arguments \rightarrow result\_type$ is constructed from the argument and result type, and added to the type set for this function.

To illustrate the use of function types, consider the following, simple program:

```
<?ℓ1
  function id^ℓ2($x^ℓ3) {^ℓ4
    return $x^ℓ5;^ℓ6
  }^ℓ7

  echo id^ℓ8,9(3^ℓ10);^ℓ11
  echo id^ℓ12,13("id"^ℓ14);^ℓ15
?>^ℓ16
```

The function `id` simply returns its argument directly as the result, which illustrates that a function can have multiple result types, depending on its input types. For example, at ℓ_9, the calculated function type for `id` is $\texttt{id} = (\{int\}) \rightarrow \{int\}$, and at ℓ_{13} we can add $\texttt{id} = (\{string\}) \rightarrow \{string\}$ to its type set.

We do not perform widening on the function types, because since a script can only contain a finite number of function calls, a function can also only have a finite number of types. However, the set of function types could grow arbitrary large, depending on recursion of the function. A careful choice for contexts and the widening operator on variable types could help reduce these type sets.

7.4.4 Adding global and static variables

The addition of global and static variables brings some extra complications to the algorithm. Static variables are preserved when leaving the function call, and their values are retained when calling the function again.

Initialization of these takes place at the first call of the function. Global variables however are copied from the global (main) scope of the program, and after the function call, the value they have at the end of this function, is copied back to the global scope.

While it may be desirable to keep track of all global and static variables at every program point (and context), usually this will mean that we are dragging extra (and often useless) information along during the fix-point iteration, because these global and static variables often will not change type. A better solution would be to keep a separate type environment for all global variables, and every function with their static variables, and update these each time such a variable is changed. However, this will require that after each change, the new type of the global (or static) variable will have to be unified with the types that have already been calculated[6].

For static variables this only concerns the function itself, but for global variables it concerns the *entire* program. Lastly, to keep our definition of global and static variables sound, each edge leading to a statement using this specific global (or static) variable that has been changed, has to be re-analysed, because we do not know where the change actually happened. But because changes to a global variable should not occur very often, this seems acceptable.

For handling global and static variables, two new functions have to be defined:

$globals : TEnv$
$statics : f \rightarrow TEnv$, where $f \in FunctionId$

The changes in the algorithm will only take place in step 2; the rest of the algorithm stays unchanged. In Figure 7.15, the changes for the algorithm to handle global and static variables are presented: the beginning of the iteration step is unchanged, except that before $exit'$ is calculated, $exit$ is updated with the global and static variable types. If *node* is on the global scope (i.e. not a node that appears inside a function), the global type set $globals$ is simply the newly calculated type set. However, if *node* is a call return node, the exit set still has to be unified with the global variables from the function exit node. If *node* is not on the global scope, then the global set and static variable set for this specific function need to be unified with the global and static variables of this function.

Then, the edges that use the changed global and static variables are added to the worklist[7]. Lastly, the exit set is unified as well, with these global and static variables. Afterward, the algorithm proceeds normally.

7.5 Results

The resulting function of the worklist algorithm contains three type sets for each identifier (variable or function), which represent the actual types of these identifiers, before and after analysis of this program point (the entry and exit sets), and the expected types. These sets are used to produce the actual result for the type inference system: warning messages that indicate any of the typing problems we introduced in the previous chapters.

Not all warnings should be treated with the same severity. For example, a coercion from a string to an integer should be treated as less severe than a coercion from an array to an integer: the string to integer coercion might succeed, while the array to integer coercion will always yield an erroneous execution[8]. Other situations for which a warning message may be generated include for example multiple types for a single variable, undefined functions and type changes for variables. We measure the severity of a warning with a value between 0 and 1, where a value closer to 0 is considered less severe than a value near 1.

In Figure 7.16 we present a list of coercions for which warnings should be generated, and the severity of these warnings. A list of warnings for other situations is given as well.

By the definition of a soft typing system, discussed in Chapter 3, either warnings should be generated, or code changes to remove the typing problems should be introduced, or both. In our system we only generate the warnings, because some of the type problems (e.g. the multiple types for functions) cannot be solved by simple insertions into the code, and the definition of a soft typing system [7, 2, 45] states that a soft type

[6] For global variables, this would also require merging on the global scope itself, when a variable that has been a global somewhere, has changed.

[7] Probably the most efficient way is storing the list of global variable (either on global scope or in a function) uses during the execution of the algorithm, whenever we encounter one. Then it is only a matter of re-adding all these edges to the worklist, when the type of that variable changes.

[8] Of course, the execution will still continue, but that value is not likely to have the intended meaning.

```
...
    Step 2: Iteration
        while W ≠ nil do
            edge := W.pop();  node := edge.to;  prev := edge.from;
            context := edge.to_context;  prev_context := edge.from_context;
            (entry, exit, _) := analysis(node)(context);
            constraints := set of constraints C^node, derived from C*[E*], without expected types;
            new_set := transfer_types(entry, constraints, ∇);
            if node is a node on the global scope then
                globals := new_set;
                if node is a call return node then
                    (_, prev_exit, _) := analysis(prev)(prev_context);
                    foreach v ∈ node.function.globals do exit(v) := exit(v)∇prev_exit(v);
            else
                foreach v ∈ node.function.globals do globals(v) := globals(v)∇new_set(v);
                statics' := statics(node.function);
                foreach v ∈ node.function.statics do statics'(v) := statics'(v)∇new_set(v);
                exit := exit ⊔ globals;
                exit := exit ⊔ statics';
            foreach id ∈ exit do
                if exit(id) ≠ exit'(id) then
                    if id ∈ globals then nodes := all nodes in analysis where id appears in global context;
                    else if id ∈ statics' then nodes := all nodes in analysis where id appears in static context;
                    foreach node ∈ nodes ∧ (node', node) ∈ F do
                        foreach c ∈ analysis(node) do W.push((c, node'), (c, node));
            exit' := exit ⊔ new_set;
            ...
...
```

Figure 7.15: The changes in the algorithm to handle global and static variables

Situation	Severity	Description
int or *float* to *num* coercion	0.2	Not a real coercion, but some type information gets lost here. Since *num* is actually a type set and not a primitive type, it becomes impossible to determine which of the primitive types *int* or *float* is the actual (real) type.
float to *int* coercion	0.3	Loss of information (decimals).
string to number coercion	0.4	This coercion may fail because the string is not a real number.
bool to number coercion	0.1	A *bool* does not have real information that should be stored in a numeric value.
object to non-*object* coercion	0.4	Coercions of this type are completely meaningless.
any type to *bool* coercion	0.3	*int* to *bool* is comprehensible, but any other coercion to *bool* has no significant meaning.
array to non-*array* coercion	0.4	Loss of information, and result value has no significance.
non-*array* to *array* coercion	0.4	This coercion will always yield an empty array, and is thus quite useless.
Functions with multiple types	0.5	Normally functions should return a single result type for all input of the same types. This warning indicates that this function returns multiple different types for the same input.
Undefined function	0.9	If for a function a result type cannot be found or calculated, then it is undefined. And for these functions it is impossible to determine what will happen with its result.
Undefined variable	0.8	If for a variable a type is expected, but no concrete type is found, then it is undefined.
Assignment of *any* type to a variable	0.1	The assignment of the *any* type indicates that a real (primitive) type for this variable could not be determined.
Type change for a variable	0.5	A type change for a variable usually indicates that the behaviour or use for this variable has been changed. Usually this indicates unintended behaviour in the code.
Multiple types for a variable	0.7	This warning usually appears at a merge of two or more branches in the code. This indicates for a variable that it can have any of the specified types, but it is uncertain which is the exact type.
Assignment of *void* type	0.5	Procedures with no return statement should not assign a result to a variable. If this happens, the type of that variable is meaningless.

Figure 7.16: Warnings for type inference results

system may not refuse any program. Some of the warnings can be ignored if the programmer knows what he is doing at that location in his script. A suppression mechanism could hide such irrelevant warnings for the programmer. However, in some cases (e.g. with undefined variables or functions), something really is wrong, and action should be taken to solve the problem.

In our system, the warnings are gathered per line in the source code. If more than a single warning message is generated for a line of code, then these are grouped, while their priorities are added together in such a way, that additional warnings strengthen the priority of already existing ones, while their total will never exceed the maximum value of 1. For example, if we have a warning message with priority 0.3 and another message with priority 0.5 is found, then their total priority would be $0.3 + (1.0 - 0.5) * 0.5 = 0.55$.

7.6 Future work and conclusions

While this type inference system is quite powerful, there are still enough points that could be improved, or enhanced. There are also points that go beyond type inference, and require additional analyses to become effective. In this section, we discuss a few of these possible improvements.

- Currently we add edges to the worklist per context, and so a node can only be processed for one context at the same time. But, the algorithm could work much faster if *all* contexts for this node are processed in parallel. However, not for all contexts analysis has to be done, so it is required to keep track of which contexts have changes and which have not. This could be done with a *dirty list*. Instead of specifying the context of the node we are analysing, we give a list of contexts instead, for which iteration has to be performed. Only the contexts in this dirty list are then considered.
- Currently, function types are not unified with each other, and each instance with different input and result types, appear in the type set calculated for this function. However, in some cases (e.g. the `id` function we illustrated in Section 7.10) it makes more sense to group all types together as one, using *type variables*. In the `id` example of Section 7.10, this could lead to a type $(\{\tau\}) \rightarrow \{\tau\}$ as result, instead of adding a new entry for each newly encountered result. However, this requires careful analysis of the function types, and this substitution could fail in cases where the expected behaviour does not hold (e.g. if the `id` function would return a *string* for an input variable of type *bool*).
- When a function f has been analysed, and for arguments $e_\star$ the types $\tau_\star$ have been found, then it should normally always yield the exact same result type, and this means re-analysis of this function for these input types for this function f is not necessary, saving a lot of execution time for the algorithm. However, static and global variables play a role too, and these may have an impact on the result type of f as well. Further research is required to determine that always the same result type is returned, when all input parameters (including statics and globals) are unchanged, for two calls of the same function f.
- Functions may have side effects that go beyond type checking. For example, the `unset` function erases a variable, while type checking will see this as a normal function call, leaving the affected variable intact. The same can be said about functions like `die` and its alias `exit`, which terminate the entire script execution. However, such side-effects cannot be predicted when the function with this behaviour is a library function for which no code is available. Thus, for each specific function, additional information about these side effects should be added and processed, complicating our algorithm.
- In our type system, callback parameters for functions are treated as strings. However, it is possible to write functions that use their own callback parameters, and our type system cannot cope with these, because they are represented by strings.
- We have not implemented PHP classes in our system. However, if this type system is to be really useful, classes should become an integral part of the type inference. Every professional library, for example PEAR [23], makes use of classes. Implementing support for classes should not be very hard either. A *class* type for class declarations should be added, with annotations for each of the class types this class may have to handle inheritance, and types for each of its members. We already have an *object* type, which should have these annotations as well. However, things are not as straightforward as they look, because classes may too have side effects, and exhibit unpredictable behaviour that impacts our type system, but cannot be detected by it.
- Some constructs, like the `try ... catch` are not supported. Adding these would give more support for more PHP programs, but would further complicate the type system as well. Also the `try ... catch` requires the support of classes.

- Conditional constraints could be introduced, so that more specific type information can be given in some specific situations. For example in the `if ... else` construct, more specific type information can be given if the exact value of the condition is known. This would also solve the multiple types problem at the merge. By adding conditional constraints, that give conditional types, it would become possible to see why and when a variable would get its type. Another example is the `switch` construct. For each of its `case` labels, conditional constraints can be used to push the case label's value into the `case` statement. This however only works if the `switch` condition is a concrete variable, and not an expression.

Even though these improvements would make the type inference system a lot more useful, it is already quite powerful, because it can handle all of the most commonly used PHP features, and gives quite precise results for many programs (see Chapter 10). Also, the type system can be used as a template for building a type inference system for similar dynamically typed languages. Using the constraint based approach that gives two kinds of results (types and expected types), any unusual coercion can be quickly detected using these two results.

Chapter 8

Analyses for PHP

In Chapter 4 and Chapter 5 we have discussed HTML validation methods and coding practices that can be applied on programs written in scripting languages, with the goal of improving the quality of its source code. In this chapter we will describe some analyses that perform the HTML validation and check if specific coding practices and protocols such as database connections are respected. For these analyses we give a definition, to which parts of the source code they should be applied, a description that further explains the reasoning behind the analysis, illustrated by code examples, and finally the feedback that is generated by the analysis. These are warning messages that indicate any problems that have been found. We also define a minimum level of detail, which we discuss below.

Analyses can be implemented at several levels of detail. The simplest form of analysis is simply applying *regular expressions*[21] on the source code directly, to pattern match on possible suspicious parts of the script's code[1]. However, such a regular expression pattern matching analysis does not respect the structure of the scripting language (e.g. branching and loops), which means that some details may get overlooked, or simply misinterpreted. A more elaborate analysis can be done when the language structure is respected, i.e. by inspection of the abstract syntax tree. This is called a *syntax-directed analysis*, because such an analysis is performed on the syntax of the language. Even more detail can be achieved if also the *execution flow* of the script is considered. This typically gives rise to a *fix-point analysis*. The type inference algorithm that we have already discussed in Chapter 7 is a good example of this.

Regular expression and syntax-directed analyses are typically applied on the entire PHP script, separately on each of the functions that have been defined in the script, and any of its included files. For most of our syntax-directed analyses we do not analyse the functions that are called, when we encounter a call to these. Instead, we analyse these functions separately. Lastly, fix-point analyses are only applied on the main program flow of the entire script, and not on the local flow of each function separately. However, we do perform analysis on these functions whenever a call to these is encountered.

In this chapter we describe several analyses, but we do not give an algorithm for any of them, because they can be implemented in several ways. Our definitions however will give a good insight into what each of these analyses want to achieve, and what kind of feedback (i.e. warning messages) can be expected as a result.

We grouped our analyses by subject. First we discuss the HTML validation related analyses. Next, we discuss the type inference related analyses. The type inference algorithm itself is already discussed in Chapter 7. Next, we discuss protocol-related analyses (database, files, sessions, etc.). Lastly, we discuss analyses that enforce a certain coding style.

[1] We assume here that the regular expressions are applied on the source code with *all* of its comments removed. These could severely compromise the reliability of the regular expression analysis, if left intact. This can be achieved by parsing the source code, and using a pretty-printed AST in the regular expression analysis.

8.1 HTML validation

1. **Missing closing tags**

   Definition For each encountered opening tag, a corresponding closing tag must exist as well.

   Minimum detail level Regular expressions, but syntax-directed analysis is more accurate.

   Applicability This analysis should be applied on every HTML generating code fragment. For regular expressions this means the source code of the main program *and* of each function separately, and for syntax-directed analysis this means every literal string that is encountered.

   Description All HTML opening tags found in a PHP code block must also have a closing tag for each of these opening tags. This means, that all tags that do not end with `/>`[2], are considered as regular tags, and a closing tag for these will be expected. The difference between regular expression and syntax-directed analysis is, that syntax-directed is more accurate. An example:

   ```
   $x = "<b><i>Test";
   if ($k < 10) $x .= "</i></b>"; else $x .= "2</i></b>";
   ```

   When executed, this example would return correct HTML, but a regular expression analysis will assume that the closing tags are printed twice, while a syntax-directed analysis recognizes the `if ... else` construct, and finds that both close the HTML tags correctly. A different example is when the `if ... else` construct is used to choose between the opening and closing tag:

   ```
   if ($k > 10) print("<ol>"); else print("<ul>");
   ... //many printing statements using <li> statements
   if ($l == 3) print("</ol>"); else print("</ul>");
   ```

   The difficulty here is that the conditions of both `if ... else` statements are different, and they concern two different variables. A syntax-directed analysis might be able to recognize a situation such as this, but because no information about the variable's values is known, syntax-directed analysis cannot make a viable decision based on the expressions. And even if both the conditions were exactly the same, the variable `$k` might have been changed by one or more of the statements between both `if ... else` statements, so it cannot even be guaranteed that the `$k` in the second condition is the same as in the first condition. As a result, the wrong closing tag *might* be printed, but the analysis cannot decide on this. Even so, both regular expressions and syntax directed analysis would find no missing closing tag in this example.

   Also, this analysis should check if singleton tags such as `<br>` and `<input>` appear with opening and closing tags (e.g. `<br> .... </br>`), which is not allowed in XHTML.

   Results For missing closing tags a warning message must be generated, as well as for non-corresponding closing tags. Also, if more closing tags are found than are opened, then these have to be reported with warnings. If a closing tag is encountered that is different from the opening tag, but both are very much alike (e.g. a `<td>` opening tag with a `</th>` closing tag), then a warning is generated also, but this opening tag *will* be considered as closed, so that more fundamental errors can be found accurately, without having the analysis to bother with two similar tags being mixed up with each other.

2. **Non-standard tags**

   Definition Check for every tag that it is a valid tag specified by the W3C standard for XHTML 1.0 Strict.

   Minimum detail level Regular expressions only; syntax-directed analysis would not add any extra information.

   Applicability This analysis should be applied on every HTML generating code fragment. For regular expression analysis this would mean the source code of the main program *and* of each function separately, and for syntax-directed analysis this would mean every literal string that is encountered.

   Requires A list of all valid tags.

[2] Tags ending with / > are singleton tags that do not have a closing tag (i.e. they close themselves).

Description For the browsers that were used roughly between 1995 and 2002, several non-standard tags were introduced, that were only supported by (some of) these specific browsers. But, with the introduction of XHTML 1.0 Strict, these tags were not included. This analysis should detect all such tags, and report them. For example, this PHP code fragment contains the non-standard `<blink>` tag, that is not part of XHTML 1.0 Strict:

```
$x = "Blinking text";
print("<blink> <B>$x</B> </blink>");
```

Another problem in this code example is the `<B>` tag, and its corresponding closing tag. In XHTML, all tags *must* be in lowercase, which means that uppercase characters are not allowed in a HTML tag anymore.

Results For each tag that is not specified in XHTML 1.0 Strict a warning should be generated. Also if a tag name contains uppercase characters, a warning should be generated as well.

3. **Tag nesting**

Definition If a tag is opened, the first closing tag should correspond with this specific opening tag, unless this tag closes itself (i.e. ends with `/>`), or if another opening tag is encountered, which must then be closed first (before the previous opening tag).

Minimum detail level Regular expressions, but syntax-directed analysis allows more accuracy.

Applicability This analysis should be applied on every HTML generating code fragment. For regular expressions this means the source code of the main program *and* of each function separately, and for syntax-directed analysis this means every literal string that is encountered.

Requires A list of all valid tags.

Description A code fragment with the tags `<b><i>`, will expect a closing tag combination like `</i></b>`, and not `</b></i>` or anything else of that kind. The last opened tag always has to be closed first, and then the second-last, and so on. But, as we already have explained in Analysis 1, some tags may be of the same level of *structure*. For example, it is not a problem for a browser to properly display a page in which a `<th>` tag is closed with a `</td>` closing tag:

```
$contents .= "<tr><th width=50>Username: </td><td width=300>$row[2]</td></tr>";
```

However it gets worse if a structural element is missing. For example forgetting a `</table>` tag makes the validator expect that the table is closed at some point, and it will complain about incorrect tag nesting at every consequent closing tag. Moreover, this can also severely deform the page layout.

Results For each incorrectly nested tag, a warning message can be generated. However, similar to Analysis 1, a distinction must be made at structurally equal closing tags; if the found closing tag is considered as structurally equal (e.g. with the `<td>` and `<th>` case), the opening tag should be closed (but the wrong closing tag must still be reported), so that more errors can be correctly detected.

4. **Tag attributes**

Definition Check whether tags have illegal attributes or not.

Minimum detail level Regular expressions only, syntax-directed analysis does not add more details.

Applicability This analysis should be applied on every HTML generating code fragment. For regular expressions this means the source code of the main program *and* of each function separately, and for syntax-directed analysis this means every literal string that is encountered.

Requires A list of allowed attributes for each of the tags.

Description Tags may have attributes. But, not every tag allows the same attributes. This analysis checks that for each tag found, the attributes for this tag are legal, for XHTML 1.0 Strict. This example illustrates an occurrence of the `<b>` tag, where an illegal `href` attribute is specified:

```
print("<b href=\"http://www.cs.uu.nl/\">is this allowed?</b>");
```

The analysis could also be made smarter, e.g. by suggesting in the above example that possibly the `<a>` tag was meant to be used here. However, stating for a tag that an illegal attribute was used, is likely to be enough.

Results For each illegal attribute that has been found, a warning message must be generated. However, it could also be very useful to give the list of *valid* attributes for that specific tag as well.

5. **Validation of GET and POST variables**

Definition User input variables are stored in two arrays: the `$_GET` array, containing variables from a GET context (i.e. from the URL directly), and the `$_POST` array, containing variables from a web form which has been filled in by a user (i.e. the POST context). There is also the `$_COOKIE` array containing cookie variables sent by the client's browser. The `$_REQUEST` array contains all elements from these three arrays. These variables may contain tainted information, and must be untainted before they can be allowed to propagate through the script.

Minimum detail level For each occurrence of these arrays, it must be checked if it is processed by an untaint function. This is possible with regular expressions, but syntax-directed analysis is also able to detect such occurrences, and has the extra advantage that identifying such untaint functions becomes very straightforward.

Applicability This analysis should be applied where GET and POST variables are found.

Requires A prefix or the name of a function that performs the untainting of input variables.

Description Data obtained through web forms and from variables passed by the URL can be tainted, which means that these can contain malicious data, possibly with the purpose of sabotaging the site, or gaining unauthorized access to information. To sanitize such data, it must be processed by an untaint function, to remove all elements from it that can be used to compromise the site's integrity. Consider the following example:

```
$a = $_GET['test'];
$b = validate($_POST['test2']);
print("Results: $a and $b");
```

In this example, the variable `$a` gets its value from the web input variable `test` directly, and is not untainted. This should generate a warning. The input variable `test2` however is given to a function first, before it is assigned to `$b`. This function `validate` however must be known to the analysis that it is an untaint function, or otherwise a warning will be generated also – the analysis cannot know if the data is untainted if it does not know that the `validate` function performs such a task –. It would also be possible to create multiple untaint functions, each processing a different type of information (e.g. one that processes integers, one for strings, etc.). This way, a form of input validation could be performed as well, though it is easier to have a simple JavaScript (on the client-side) perform this task[3], before the web form variables are sent to the web server[4]. The following example shows another issue:

```
$b = $_POST['test2'];
$b = validate($b);
```

This example could be seen as tainted, but the `$b` is validated in the second statement. A regular expression analysis would not detect this, and with a syntax-directed analysis it would be quite complicated to detect this also. For such cases, only fix-point analysis would be a viable solution.

Results For each user input variable in a script that is not processed by an untaint function before it is used in an expression or assigned to a variable, a warning message must be generated.

8.2 Type inference related

Although the type inference analysis of Chapter 7 would detect all of the problems in this analysis also, it is a very expensive analysis, and for large scripts it could take considerable time to perform full type inference. Therefore, we have also made several smaller syntax-directed analyses that would require considerably less

[3]Moreover, JavaScript input validation is also faster.

[4]However, this still does not remove the need to do the validation at server-side also, because JavaScript can be disabled by the browser, so that client-side validation is not performed.

time. The third of these analyses (i.e. Analysis 8, assignments in expressions) explores the hazards of assignments in expressions, which can also cause problems that type inference could not find. Each of these analyses should be applied on the entire PHP code, i.e. its main program and every function.

6. **Expressions of conditional statements**

   Definition Expressions of conditional statements such as `if ... else` and `while (...) { ... }` should be boolean expressions.

   Minimum detail level This analysis requires syntax-directed analysis.

   Description Because of PHP's dynamic typing, any possible value can be used as the condition in a `if ... else` statement, for example. But, not every such value actually has some meaning as a conditional, because nearly all values are converted to `true` (see Section 6.1.7 for details) when used in a coercion to boolean value. Consider the following example:

   ```
   while ($row = mysql_fetch_row($result)) { print_r($row); }
   ```

   This example works perfectly, for as long as the `$row` variable gets a value from the `mysql_fetch_row` function, which returns a row array for as long as the query resource variable `$result` contains any, otherwise the valid `false` will be returned by this function, and this actually is a valid boolean value. The use of such an assignment in a condition is comprehensible, but what if a different function was used (with a different result)? A safer solution here would be to use `mysql_num_rows` to test on the number of rows returned by the query:

   ```
   $num_rows = mysql_num_rows($result);
   for ($i = 0; $i<$num_rows; $i++) { $row = mysql_fetch_row($result); print_r($row); }
   ```

   This code example gives exactly the same results as the first, except that no non-boolean conditions exist in the script anymore.

   Results For every conditional expression which is not of boolean type or for which it cannot be directly determined (without the need of full type inference) that it is of a boolean type, a warning message must be generated.

7. **String concatenation: consistent usage of the `+` and `.` operators**

   Description This analysis checks that expressions using the `+` operator must have numerical operands, and that expressions using the `.` operator have (preferably) string operands.

   Minimum detail level This analysis requires syntax-directed analysis.

   Description Several languages (e.g. Java) use the operator `+` for both string concatenation and addition of numeric values. In PHP however, there are two separate operators, one for numeric addition, and one for strings. Unfortunately, programmers that use both languages and are not very experienced with PHP, could get confused when using these operators, and use `+` in PHP where `.` is required. This could give problems when trying to find an error in the script, and such a misuse of the `+` operator is involved. Consider this example:

   ```
   $a = 4;
   $b = 3;
   $c = "Test1";
   $d = "Test2";
   $a = $a . $b;
   $c = $d + $b;
   $e = $a + $c . $d;
   ```

   Here, both operators have not been used correctly. First, the values of `$a` and `$b` are used in a string concatenation, while both are integers. This however would not pose any problem in PHP, since integer to string coercions are acceptable (though it does change the type of `$a` to string in the assignment statement). Next we try to add `$d` and `$b` together. Here the coercion of the string value of `$d` fails, and the neutral value `0` is chosen as its substitute. Finally, we try to add `$c . $d`[5] to the value of `$a`. However, the type of `$a` was changed to a string already, which means that another incorrect use of the `+` operator has been detected.

   Results For each incorrect use of the `+` or `.` operator, a warning message must be generated.

[5] The . operator binds more strongly than the + operator.

8. **Assignments in expressions**

   Description This analysis checks if side effects exist in expressions, of the form `$a = $b`, `$a++`, or a combination of both.

   Minimum detail level This analysis requires syntax-directed analysis.

   Description Assignments in expressions can be very useful, especially when used to assign a single value to multiple variables within a single statement. However, if badly applied it is not always clear which value is assigned to a variable when the assignment is an inner expression. A useful example of an assignment expression is discussed in Analysis 6 where it is part of a conditional. However, even this may not be desirable, because the result type of such assignments are often non-boolean, which may cause a coercion to take place that could lead to unpredictable result values. The following illustrative example shows exactly what can be deemed acceptable, and completely unacceptable:

```
$a = 1;
$b = 6;
$c = ($a = $b); //this one is defendable
$d = 4;
$e = ($a = $c) = $e++; //this does not even parse!
//this one however does, but the outcome is very dependent on execution-order:
$f = ($e++) + ($a = $b);
//and what would be the result of this??
$g = 1;
$g = ++$g + $g++;
```

   As the comments in the example state, the statement `$e = ($a = $c) = $e++` will not even parse. The last statement does, but it is uncertain what the actual value of `$f` is. The expression `$e++` is also considered as an assignment, but as it is a postfix increase, it will be executed after the full expression has been evaluated. However, if the programmer does not have such knowledge (or is unsure of it), a statement like this may lead to a lot of confusion. The last statement is even less comprehensible, unless the programmer has detailed knowledge of how the `++` prefix- and postfix operator works[6].

   A different situation would be if such an occurrence happens in an `if ... else` statement:

```
if ($a = 4) { $b = 3; } else { $b = 5; }
```

   In this situation, the `$a = 4` assignment is likely unintended, because the programmer probably wants to test if the value of `$a` is equal to `4` (i.e. he meant the expression `$a == 4`). However, instead the value `4` is assigned to variable `$a`. Because the value `4` always converts to the boolean value `true` when used in a condition, the value `3` is assigned to `$b` in the `then` clause. The `else` clause therefore is *never* even considered! Situations such as these often happen when the programmer is used to languages where the `=` operator is used for equality checks.

   Results For each assignment found in an expression, a warning message must be generated.

8.3 Protocols

We only consider protocols that have the following characteristics: first a connection to some resource is created, then several operations are performed on that resource, and then the connection to this resource is closed. We assume this kind of behaviour of protocols in all of the analyses we describe here. Sessions are an exception to this default behaviour though, because the ending of a session can not be done until the session itself is no longer needed (e.g. when a user logs out of a website). And because sessions can span multiple web-pages, it cannot be enforced that every page starts a new one – unless it is a different user (and thus a different instance of the site).

All of these analyses should be applied on the main program of the PHP script. Because functions of each protocol can also be used inside of user functions, these should be checked as well. Fix-point analysis would give the best results, but by just collecting all of the protocol functions that could be called from each

[6] The result of this statement is that `$g` gets the value 4.

function and using a syntax-directed approach, it is already possible to get a good idea if the protocol rules are respected – yet there is no guarantee that the warnings generated by the analysis will be fully accurate; there's a high risk of false positives here.

9. **Databases**

   Definition Check if the database handling protocol is respected.

   Minimum detail level Syntax directed analysis can give a good idea about the order in which the database handling functions are called. However, to check if a function that makes use of a database connection, has been called before a database connection is opened, control/data-flow analysis should be performed.

   Description PHP has support for several kinds of databases (e.g. MySQL, PostgreSQL, etc.). We will only consider MySQL in our example code. For opening a database, the function `mysql_connect` is used, followed by the `mysql_select_db` function to select the actual database. Then, several operations can be performed using the other functions, e.g. `mysql_query` to perform queries (of which the result rows are retrieved using functions like `mysql_fetch_array` or `mysql_fetch_assoc` for example), or other functions, to inspect result size (field sizes, row lengths, etc.). Finally, a the `mysql_close` function must be called to close the database connection.

   There can be several database connections (to different databases) at once in a PHP script. For each of these, a database connection *resource* is created, from which query resources can be made. If for a query no connection resource is given, the *default* connection (i.e. the oldest created database connection that is still active) is used. Consider the following example:

```
$con = mysql_connect("localhost", "dbexample") or
  die("ERROR - could not connect to the database");
$db = mysql_select_db("test") or
  die ("ERROR - requested database not present");

$id = untaint_id($_GET['thread_id');
$result = mysql_query("SELECT * FROM Threads WHERE Forum_ID = '$id');
while ($row = mysql_fetch_assoc($result)) { ... }

...

mysql_close($con);
```

   In this example, the connection is first made, and a database is selected. If an error is encountered during this process, a suitable error message will be shown (and script execution terminated). Then, an `id` value is obtained from a user input source, and untainted before being used in a query, that retrieves a set of row results from a sample database. Finally, the connection is closed.

   The behaviour in this example is correct; the database is first opened, then used, and closed afterward. Usually, the open and close actions for databases are covered by user-made functions, which makes it more straightforward to switch between database types, when it gets changed. One function call that could have been added after the `while` loop, is one call to the `mysql_free_result` function, to free the resources used by the `$result` variable. However, these actions are always automatically done by the PHP interpreter at termination of a script. But, if a single script uses many different queries (or query variables), using the `mysql_free_result` after each completely query could speed up the entire script, because this function frees all resources (including memory) that have been used by these queries.

   Results If a database query is performed before a connection is made (or after it is closed), generate a warning message. A warning should also be given if the database connection is not closed before the script terminates. Optionally, a warning can be generated if the `mysql_free_result` function is not called for each completed query. Consequently, warnings should then also be generated for query resources that are accessed after the `mysql_free_result` function has been used to free their memory resources.

10. **Sessions**

    Definition Check if the session protocol is respected.

    Minimum detail level Syntax directed analysis can give a good idea about the order in which the session functions (as well as the `$_SESSION` superglobal) are used. However, fix-point analysis should be performed, to check if the session may be closed (or destroyed) before the last use of a session variable or function.

    Description A session can be used to add state to a website. They have to be opened with the `session_start` function *before* anything is printed to the browser, otherwise the interpreter itself will throw an error. Then, several functions can be used to manipulate the session data, and the `$_SESSION` superglobal becomes available to store information. There can only be *one* session at the same time; sessions can be closed at the end of a page using the `session_write_close` function, that saves all session data, and closes it. However, it will not *destroy* the session; when another page is opened that uses sessions, the session data from the previous page will be used. To fully end a session (and destroy its data), the `session_destroy` function must be called.The following example shows an incorrect use of sessions:

    ```
    unset($_SESSION['var1']);
    unset($_SESSION['var2']);
    unset($_SESSION['var3']);
    session_start();
    $_SESSION['var4'] = $_REQUEST['var5'];
    foo($_SESSION['var5']);

    ...

    unset($_SESSION['var5']);
    ```

    Several mistakes are made in this example. First, the `$_SESSION` variable is manipulated *before* a session is started. This means, that these session variables may still exist after the `session_start()` call. Next, the web input variable `var5` is not untainted before it gets assigned to a session variable (see Analysis 12 why this is harmful). And lastly, the `session_write_close()` function is not called to close the session at the end.

    Results If a session variable is used before the session is started, generate a warning message. A warning should also be given if the `session_write_close()` function is called before the script terminates (unless the `session_destroy()` function was called).

11. **Files**

    Definition Check if the file access protocol is respected.

    Minimum detail level Regular expressions can give a good idea about the order in which the file access functions are used. Syntax directed analysis will give more precise results. However, fix-point analysis is needed, if file handles are used as function parameters, and processed by functions.

    Description A file must be opened using a call to the function `fopen`, which takes two arguments, a filename, and a parameter indicating the handle mode (read or write), and some flags that must be applied to the resource (e.g. for write mode, a flag can be added to indicate new actions should append to the file, and not overwrite it). The result of `fopen` is a file handle, which must be assigned to a variable, so that read or write actions can be performed. Reading a file is usually performed by the `fgets` function, that reads a line from a file, or the `fread` function, that can be used to read binary files. Writing to a file is done with the `fputs` or `fwrite` functions. Both the read and write functions require the file handle as a parameter in order to perform their operations. Closing the file is done with the `fclose` function. The following example illustrates reading from a file:

    ```
    $handle = fopen("/home/file.txt", "r");
    $var = array();
    while (!feof($handle)) {
      $var[] = explode(" ",fgets($handle,4096));
    }
    fclose($handle);
    ```

This example opens a file called `file.txt`, and then reads it line by line (while checking if the end is reached using the `feof` function) using the `fgets` function. These lines are then split into substrings using the `explode` function, where spaces are used as the delimiters. These are then collected in the `$var` variable. Finally, the file gets closed by the `fclose` function.

Results If a file is read from or written to before it is opened, generate a warning message. A warning should also be given if the file does not get closed before the script terminates. Additional warnings must be given if an operation on a file is performed in the wrong context (i.e. a read action on a file opened for writing, and vice versa).

8.4 Coding styles

In this section we describe analyses for the coding styles that we discussed in Section 5.4. Most of these analyses are very straightforward, and do not require a code example to be properly explained. For other analyses we could give code examples, but these would not add anything useful to the description. Lastly, the first analysis we describe, the Tainted data analysis, is an extension of Analysis 5 which has already been well-illustrated. All of these analyses should be performed on the whole script – i.e. the main program, and each function separately if regular expressions or syntax-directed analysis is used, or the main program flow if fix-point analysis is used.

12. **Tainted data**

    Definition Check if data obtained from any external source other than the script's own variables, is properly untainted before it is allowed to propagate through the script.

    Minimum detail level It is possible to perform most of the taint checks using syntax-directed analysis. However, fix-point analysis can be useful to learn how far tainted data can spread through a script, and what the consequences are, resulting in more useful feedback.

    Requires A prefix or the name of a function that performs the untainting of data.

    Description As we described in Analysis 5, web input data can be tainted, and need to be carefully checked before being allowed to propagate through the script. This analysis should not only perform the validation of web form data (i.e. the analysis described in Analysis 5), but also validation of other external input sources, such as sockets, files (local as well as remote), and everything else that can input new information on the website.

    Basically the same approach as for web form input validation is taken: check if the data is untainted by an untaint function (either user-defined or from a library), and report if this is not the case. However, some data can be considered less tainted (e.g. local files), if we can assume that this data is safe. For local files we can make this assumption, if these are generated by the site itself, from non-tainted data. A more elaborate description and many examples of untainting of data can be found in the PHP-Sat tool [4], which is designed to identify and safely eliminate many security issues in PHP source code, and also the work of Huang et al. [29, 28].

    Results For each occurrence of tainted data that is not untainted by an untaint function, generate a warning message. If fix-point analysis is used, more sensible warnings could be generated, depending on the nature and spreading of the tainted data.

13. **Printing statements**

    Definition Check if one style of printing statements is used, or multiple styles.

    Minimum detail level This analysis can be fully performed using regular expressions.

    Description It is often considered tidy to have only one kind of printing style for an entire script, instead of mixing the use of the `echo` statement and `print` function constantly, especially because both can achieve the same results, though `echo` allows multiple arguments (`print` can only handle these, if they are concatenated using the `.` operator). In general, `echo` is the mostly used printing method, because it seems to run slightly faster than `print` does.

    There are also other functions that can print information to the browser. For example, the `print_r` function prints text like the `print` function does, but in a human-readable format. Arrays for example are very readable when printed with `print_r`. The `printf` function prints text also,

but uses a formatter string, which can be used to format strings and numbers more nicely. This function originates from the C language.

Lastly, another style that can be used is by simply gathering all text that has to be printed to the browser in one or more variables, and at the end concatenate them, and print the result with one printing statement. The advantage here is, that the text can be validated before it is actually printed, which is e.g. useful for HTML validation from within PHP code itself. It is also useful when the page is formatted using a default template, that determines where on the page each string must be printed.

Results If more than one style of printing statement is used, generate a warning for each occurrence of the least appearing printing statements.

14. **Code blocks**

Description This analysis checks that no code block exceeds a given maximum amount of statements.

Minimum detail level This analysis can be fully performed using a syntax-directed approach.

Requires A parameter that determines whether a code block can be considered as "too large".

Description Large code blocks are often very unreadable, especially if these code blocks are badly structured. It is better to split these code blocks into several functions, that together perform the tasks of the entire large code block. The size of a code block can be measured in several ways, of which the total number of statements is the simplest choice. Usually, a size of 20 to 30 statements is considered as a decent maximum for a code block.

Results Generate a warning message for each code block that is considered as too large, where for larger code blocks the priority should be higher, than for smaller ones.

15. **Single-quoted strings**

Definition Check for the use of single-quoted strings in expressions.

Minimum detail level This analysis can be fully performed using a syntax-directed approach.

Description Single-quoted strings are usually not needed, because their functionality can also be implemented using double-quoted strings, which also allow interpolation. Using one kind of strings avoids confusion, especially when the single-quoted strings contain dollar signs or curly braces, which are used in double-quoted strings for interpolation of variables and function calls respectively. Besides, these characters could be displayed with double-quoted strings by escaping them with backslashes.

Results Generate a warning message for any encountered single-quoted string. If the single-quoted string also contains dollar-signs (which indicate the intended use of variables), or curly braces (indicating the intended use of function calls or complex array expressions), generate additional warning messages.

16. **Behaviour of include files**

Definition Validate the behaviour of include files, by a set of given parameters.

Minimum detail level This analysis can be fully performed using a syntax-directed approach.

Applicability This analysis should be applied on all encountered include files.

Requires A set of parameters that determine what an include file is allowed to contain, e.g. *"to allow to contain statements that are executed when the file is included"*, or *"to disallow to contain certain kinds of statements"*, etc.

Description Include files often are only used to contain functions that are used by several other pages of the website. Also, they sometimes contain statements that need to be executed very often. However, it is safer to put these statements in a function as well, and call that function whenever these statements have to be executed.

Include files sometimes also include other files. However, these have to be handled with care, because these include files may include other files again, which could eventually lead to a cyclic inclusion of files. Another point that requires extra care is inclusion of files inside of a function.

Especially when the included file contains other functions, which are then only accessible from within the function where the include statement is located.

Function inclusion is preferably done on top-level scope. Including files within functions inside functions is definitely undesirable. Executable statements in an include file are usually undesirable also, but e.g. these statements could be constant definitions, which are used everywhere throughout a script, and moving these to a function will have the result that these constants may no longer work.

Results Generate a warning message if disallowed behaviour is detected in an include file.

17. **Deprecation of identifiers**

Definition Check if deprecated names of library functions, variables and constants are used.

Minimum detail level This analysis can be fully performed using regular expressions. The analysis is performed on a pretty-printed version of the parsed source code (i.e. without any of the comments).

Requires A list of deprecated names.

Description In PHP several names have been changed during its development, making old names (that still exist for the sake of backward compatibility) deprecated. Because no guarantee exists that these names still exist in future PHP versions, they should be reported, and alternatives offered wherever possible. Several examples of deprecated identifiers exist, e.g. the `$HTTP_GET_VARS` array, which has been replaced by the `$_GET` array. It is also important to know that although both arrays have the same meaning, they are actually syntactically different arrays. This means that changing a value in such an array, will not change the corresponding value for the other.

Some functions and constants have also been replaced by new ones, or are even discarded entirely. Especially this last group should be checked carefully, because if these functions and constants (that have been discarded) are not removed, a script may no longer function properly when a new version of PHP is used.

Results Generate a warning message if a deprecated name is detected, and if available, give the alternative (valid) name.

18. **Names of global variables**

Definition Check if variable names that are used as global variables in some functions, are used as non-global variables in other functions.

Minimum detail level This analysis can be fully performed using regular expressions.

Description Global variables that are commonly used, should have unique names for the entire script. If another function uses the exact name of a commonly used global variable for a locally used variable, the local (non-global) variable can be mistaken for the global one. Also, if the function gets changed, and its new definition actually uses this global variable, all the occurrences of the variable name have to be renamed, because otherwise there might be a name conflict, resulting in the loss of value of the previously local variable.

The same can be said for static variables, but these are only specified per single function. However, a programmer must be careful with static variable names also: these also should be different from commonly used global variable names (for the same reasons as with local variables).

Results Generate a warning message if a possible name conflict is detected.

8.5 Conclusion

Several more analyses could be thought of, but the ones that we described cover the HTML validation, protocols and coding practices that we discussed in Chapters 4 and 5. Some of these analyses seem quite trivial and straightforward, yet they are very useful in finding a lot of mistakes, and vulnerabilities in PHP programs. Also, the specified levels of detail can be very useful for deciding the amount of time one wishes to spend on an analysis, and the desired accuracy of it, because lower levels of detail could run much faster

on larger scripts[7]. Lastly, because we use a pretty-printed version of the source code for all of the regular expression analyses, the total execution time would not improve very much compared with syntax-directed analyses, for very small scripts. However, the added advantage of using these pretty-printed versions, is that we are not hindered by comments in the original source code.

To validate our analyses, we have built a prototype tool, which implements the type inference algorithm that we described in Chapter 7 and some of the analyses described in this chapter. In Chapter 9, we discuss our implementation and architecture of the tool, and discuss the software that we used in its development. In Chapter 10 we will discuss how our prototype works in practice, by testing it on several sets of PHP programs of a PHP programming assignment, and also on a website written by this thesis' author.

[7] Tests showed that the regular expressions ran about twice as fast as syntax-directed analyses performing the same checks. This is excluded parsing time of the source code though.

Chapter 9

Implementation

In Chapter 7 we described a type inference algorithm, followed in Chapter 8 by several analyses that can be performed on PHP code to detect various kinds of programming mistakes. In this chapter, we describe a prototype tool, PHP Validator, that implements these analyses. Our tool is set up in such a way, that it can be extended to work for similar (scripting) languages. Our analyses however are implemented for the PHP language only.

Our tool has been written in Java 1.5, because this is a widely used language, and this way, our tool can be run on many different platforms, because of Java's platform independence. This also brings the advantage that our tool can be integrated in an IDE such as Eclipse, where it could run as a plug-in for a PHP editor. The reason for choosing for the Java language was mainly decided by the availability of a working parser for PHP with position information[1], and our own preference for and expertise with the Java language.

First, we give an overview of the main framework, and its package structure in the Java code. Next, we discuss the architecture for the several kinds of analyses that we support. Then we discuss the interface of the tool, and finally we discuss how new languages and analyses can be instantiated in our framework. Our implementation of the PHP analyses will serve as an example.

9.1 Main framework

Our tool is structured in such a way, that adding new languages to the framework would not affect the analysis architecture in any way. Our framework basically consists of two parts: one part that contains the main infrastructure that all analyses require, as well as abstract classes and interfaces that need to be supported by the AST structure for each language, in order to identify the several types of AST nodes that each (kind of) analysis uses.

The second part contains all the classes and packages required for the languages that are analysed. This includes the parser, the AST structure, some language specific container classes and, obviously, the analyses themselves.

The main framework is organised in four packages.

`framework` The main package, containing classes for the analysis engine, and for displaying debug information.

`framework.analysis` This package contains the top-level classes of the analysis architecture, and some sub-packages with support classes.

`framework.exception` This package contains exception classes that can be used by all analyses.

`framework.util` This package contains utility classes that give support for control-flow representations, position information, and some widgets to represent pairs and triples of objects.

The `framework.analysis` has two sub-packages: `framework.analysis.ast`, which contains the main abstract class that *all* AST nodes must extend, and several interfaces, that every AST class with specific characteristics

[1] The parser we used however was very limited and required many changes to support all of the PHP functionality that we investigated with our analyses.

Class	Description
`Node`	The most basic representation of an AST node. The `AbstractAstNode` class implements this interface.
`AbstractAstNode`	The superclass for AST nodes of any language. It provides the basic requirements for all AST nodes.
`CallNode`	This interface models the start node of a function call.
`CallEndNode`	This interface models the return node from a function call.
`FunctionNode`	This interface models the entry node of a function.
`FunctionEndNode`	This interface models the exit node of a function.
`CompoundNode`	This interface must be implemented by all nodes that have references to children nodes.
`STNode`	This interface is used by syntax-directed analyses to identify (sub)programs (i.e. the main program, but also subprograms that are included by include statements).
`CompoundStmtNode`	This interface must be implemented by AST classes that model compound statements (i.e. a sequence of statements).
`IncludeNode`	This interface models an include statement.
`SDNode`	This interface is used by syntax-directed analyses to define nodes that have references to inner statements. The semantic meaning of this is similar to that of `CompoundNode`, but concerns specifically statements here (and not also expressions).
`SkipNode`	This interface must be implemented by AST nodes that must be skipped (i.e. have the identity transfer function) in fix-point analyses.
`ProgramNode`	This interface models the root node (i.e. start node) of a whole program.

Figure 9.1: The list of classes and interfaces defined in package `framework.analysis.ast`.

must implement. Figure 9.1 lists the definition of each of these classes and interfaces. The other sub-package, `framework.analysis.cf`, contains support classes for control-flow analyses.

9.2 Analysis architecture

In this section we take a closer look at the analysis architecture, and its class tree. Figure 9.2 displays an overview of the class hierarchy of our analysis architecture.

The `framework.analysis.AbstractAnalysis` class is the root class for all of the analyses, containing methods that are required for all analyses. It defines the abstract `makeResults()` method, that all analyses require to implement, and it typically initializes all analysis tasks. Also, methods exist in this class that allow finding nodes of a given type (i.e. of a given class name), and for generating warning messages. In the next subsections we discuss implementation details for each of the types of analyses.

9.2.1 Regular expression analyses

The `framework.analysis.AbstractRegExAnalysis` class is the root class for all regular expression analyses. It contains the method `findOccurrences()` which takes a node (or a list of nodes) and a regexp pattern, and searches for matches in the source code for this (list of) node(s). The result is a list of `RegExpResult` objects, each containing the location of the match, and the matched item.

For regular expression analyses, only the `makeResults()` method is required, because the implementation of a regular expression analysis can be done in many different ways. The standard approach is to apply a regular expression on the main program, and on each function separately, and then do some post-processing on the results. This post-processing can for example be the checking whether a given pattern is followed by another, checking if certain expected patterns are missing, etc. The regular expressions that we have used are defined in the `framework.analysis.RegExp` class. The pattern matching is done with the `findOccurrences()` method, that takes an AST node and a regexp string.

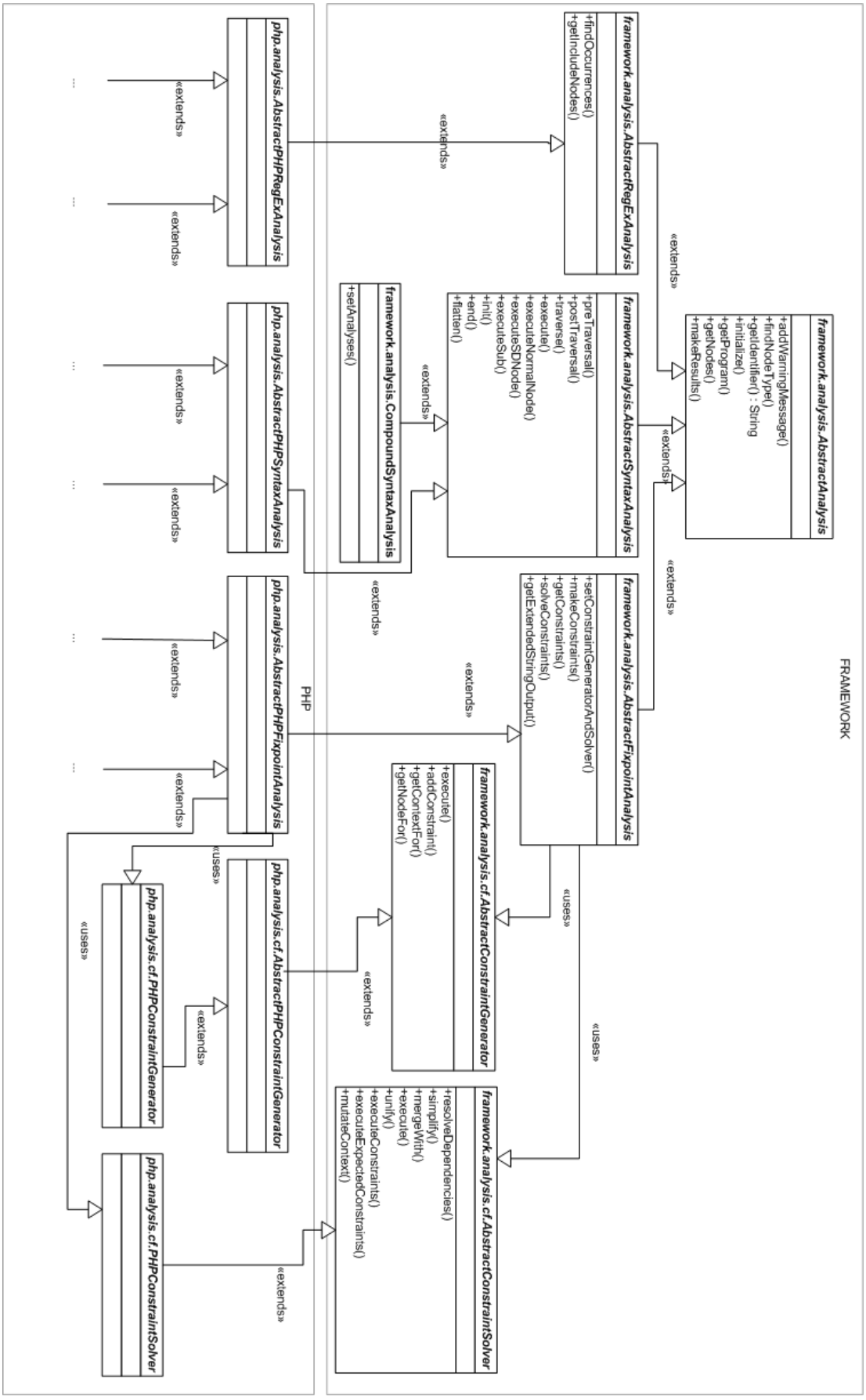


Figure 9.2: An overview of the analysis architecture.

9.2.2 Syntax-directed analyses

The `framework.analysis.AbstractSyntaxAnalysis` class is the root class for all syntax-directed analyses. It contains all methods that are required to traverse an abstract syntax tree. The most important method is `traverse()`, which is used to start the traversal. Usually this method will only contain a call to the `execute()` method, which starts the traversal at the root node of the AST. The other (abstract) methods in this class are used to define the actions for each node (`executeNormalNode`), each compound node (i.e. each node that contains branches to lists of other statements) that has been encountered (`executeSDNode`), and pre- and post-traversal actions to perform some pre-processing of the AST and post-processing of the results.

For a syntax directed analysis we do not have to define the `makeResults()` method. Only the traversal methods need to be specified. If the analysis requires it, `preTraversal()` can be used for tasks that need to be done before the traversal, and `postTraversal()` to specify post-processing tasks.

Composition

The `framework.analysis.CompoundSyntaxAnalysis` class is a direct subclass of `AbstractSyntaxAnalysis`, and it serves a special task in the framework: it is used to run multiple syntax-directed analyses on the same AST in a single pass, instead of running each analysis separately. The advantage of using this approach is that large scripts can be analysed much faster, because only one pass over the AST is required instead of multiple ones. The use of this class is automatic; the analysis engine will automatically create an instance of the class when multiple syntax-directed analyses are requested by the tool user.

9.2.3 Fix-point analyses

The `framework.analysis.AbstractFixpointAnalysis` class is the root class for all fix-point analyses. This class contains methods that initialize the set of constraints that are used in the fix-point analysis, and methods for starting the constraint solver (i.e. the fix-point iteration).

For building a fix-point analysis in our framework, only the constraint generator class (which has to be a subclass of the `framework.analysis.cf.AbstractConstraintGenerator` class), and the class containing the transfer function (which must be a subclass of `framework.analysis.cf.AbstractConstraintSolver`) need to be specified. For the PHP analyses, we have provided subclasses of these, that provide (empty) methods which are used for generating the constraints. These methods have names such as `makeConstraintsForBoolLit()`, which can be used to specify constraints for that specific type of node. Only the methods for the node types for which constraints should be generated, have to be overridden.

9.3 Interface

We decided to keep the actual interface of our tool as minimal as possible, so that it can be easily integrated into any PHP programming environment. As a result, the output of our analyses is simply a list of warnings, that can be interpreted and presented to the user in whatever way is suitable.

The following code illustrates how the tool needs to be started:

```
AnalysisEngine engine = new AnalysisEngine();
engine.analyseFile(new String[]{"FP:TYPE_INFERENCE!","SD:HTML_ANALYSIS"},"examples/");
```

As shown in the example, the filename does not have to be a single file; a directory can be specified instead, in which case all of the `.php` and `.inc` files found in this directory and its sub-directories will be analysed. Instead of a single file or directory, an array of files or directories can be specified as well. The list of analyses is specified by an array of analysis identifiers (as strings).

It is also possible to use a configuration file for running the tool. This configuration file uses a specific format and syntax, which is explained in Appendix B.2. The following code illustrates how the tool is started in Java using a configuration file:

```
AnalysisEngine engine = new AnalysisEngine();
engine.analyseFromConfig("data/analysis-example.txt");
```

The results of the analyses are grouped in an ordered map, which contains lists of warnings for each line of the script. These warning messages can be printed, or displayed in a table, depending on the preference of the user. Formatting the output of these messages is left to the user of the tool.

A list of warning messages generated by our tool could be displayed as follows:

```
case1/userinfo.php:33 Priority: 0.271
Class: FP:TYPE_INFERENCE - NEW_VARIABLE
Message: Non-concrete type assigned to variable $result_topics
Found: $result_topics=[any]

Class: FP:TYPE_INFERENCE - NEW_VARIABLE
Message: Non-concrete type assigned to variable $row
Found: $row=[any]

Class: FP:TYPE_INFERENCE - NEW_VARIABLE
Message: Non-concrete type assigned to variable $result_posts
Found: $result_posts=[any]
```

This example shows three warning messages that all occur on a single line in a PHP script. The priorities of these messages are combined, to give this line of code a higher priority than each of the messages alone. The warning classes and their priorities are explained in Appendix A.

9.4 Instantiating the framework

In the introduction of this chapter we mentioned that this framework can be instantiated for more languages than just PHP. We have only specified analyses for the PHP language, but with little effort, it is possible to also make these analyses work with other languages, e.g. Perl. Adding new analyses to existing languages is possible as well. We discuss in this section how to make an instantiation from our framework for PHP, and how analyses can be added to this instantiation.

9.4.1 PHP

To add a new language, several tasks need to be performed. First, a parser for the specific language must be written, which must also provide position information for each of the nodes in the script's source code, and a pretty printer that can pretty print the AST. The classes representing the abstract syntax tree must also implement the appropriate instances for each of the node types that were specified in Figure 9.1. Finally, the last step is to specify the analyses on this language. In this section we describe our instantiation of the framework for PHP.

In Figure 9.3 an overview of the framework packages is displayed, and their relation with the PHP packages. Like the framework itself, the PHP packages are divided into several groups, each with a different purpose.

`php.analysis` This package contains all analysis classes, grouped into sub-packages by subject.

`php.ast` This package contains all of the AST classes needed to represent an abstract syntax tree for a PHP script.

`php.parser` This package contains the parser for PHP, and several support packages and classes.

`php.util` This package contains several utility classes for our PHP analyses.

The `php.analysis` package contains the PHP-specific instantiations for the three main analysis classes that were defined in `framework.analysis`. The `php.analysis.cf` contains the PHP specific instantiations for the classes defined in the `framework.analysis.cf` package. The other sub-packages contain concrete implementations of our analyses on PHP, grouped per subject. More details of these analyses can be found in Chapter 8, and the details about the generated feedback (warning messages) in Appendix A.

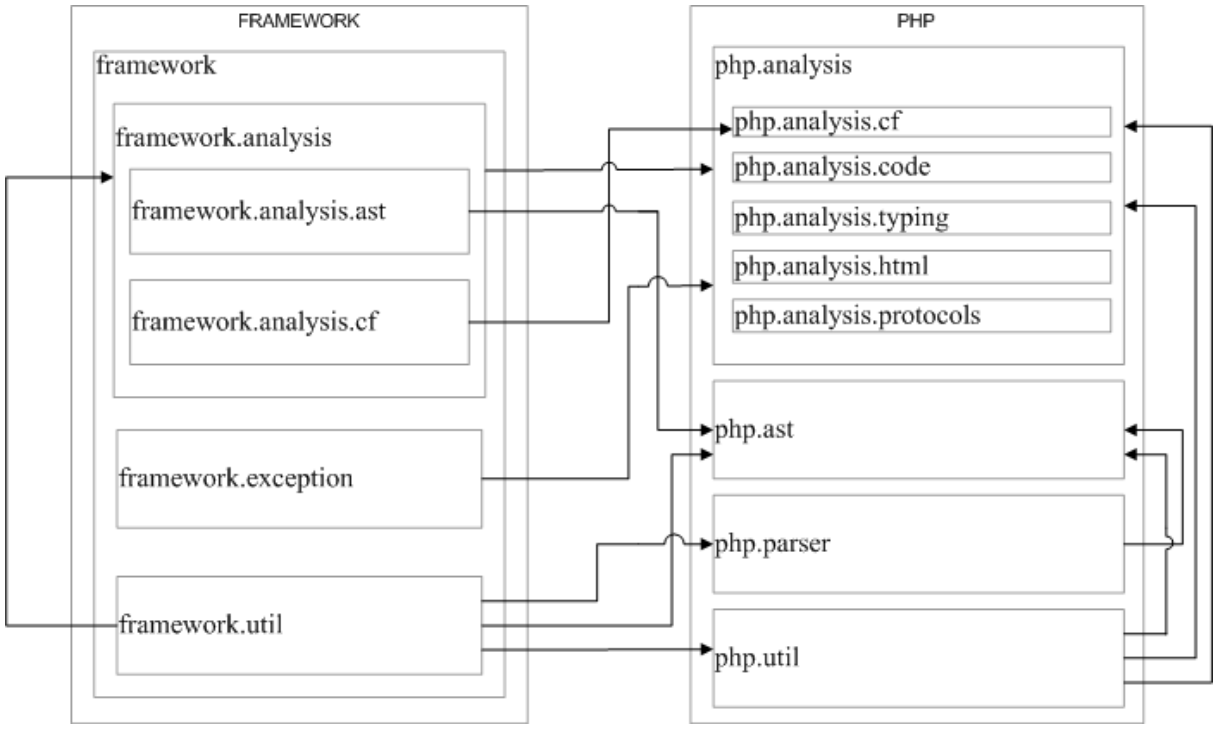


Figure 9.3: An overview of the framework and the PHP packages. The arrows indicate that a package is being used (i.e. classes being extended or interfaces being implemented) by the other package.

PHP parser

The PHP parser has been implemented using JavaCC [40], a parser generator for Java. Our parser applies all of the restrictions that we impose on PHP (these have been described in Section 6.2).This parser generates a concrete parse tree (represented by the classes in the `php.parser.cpt` package), which is used for the construction of the AST. Using a few pre-processing steps – performed by the classes in the `php.parser.cpt_to_ast` package – several problems such as include files that cannot be found (or derived directly, if not specified with an atomic string value), redefinition of functions, and some of the restrictions described in Section 6.2 that cannot be imposed by the parser itself directly. This parser generates exact position information for each node in the AST, which is used in the feedback generated by the analysis tool.

Analyses

Our analyses for PHP are all implemented in the `php.analysis` package, and their sub-packages, sorted on subject.

`php.analysis.code` This package contains all analyses related to coding styles.

`php.analysis.html` This package contains all analyses related to HTML validation using the script's source code.

`php.analysis.protocols` This package contains all analyses related to correct handling of protocols.

`php.analysis.typing` This package contains all analyses related to type inference. This includes e.g. simple checks on types of conditionals, but also the whole type inference algorithm of Chapter 7 (with its sub-packages) can be found in this package.

We have not implemented all of the analyses discussed in Chapter 8. To use the implemented analyses in our framework, each of them has been given an identifier, to which the tool user can refer to when invoking the analysis tool (see Section 9.3 for an example). A list of analysis identifiers for analyses that we have

Id	Class	Type	Analyses
RE:HTML_ANALYSIS	php.analysis.html.HtmlRegEx	Regular expressions	1, 3
RE:HTML_NONSTANDARD	php.analysis.html.HtmlNonStandardRegEx	Regular expressions	2
RE:HTML_ATTRIBUTES	php.analysis.html.TagAttributesRegEx	Regular expressions	4
RE:CODE_GLOBAL_NAMES	php.analysis.code.GlobalNamesRegEx	Regular expressions	18
SD:HTML_ANALYSIS	php.analysis.html.HtmlSD	Syntax-directed	1, 3
SD:TYPE_CONDITIONS	php.analysis.typing.TypeConditions	Syntax-directed	6
SD:TYPE_CONCAT	php.analysis.typing.TypeConcat	Syntax-directed	7
SD:TYPE_ASSIGNS	php.analysis.typing.TypeAssigns	Syntax-directed	8
SD:TAINT_GET_POST	php.analysis.code.GetPostTaint	Syntax-directed	5
SD:CODE_BLOCKSIZE	php.analysis.code.BlockSize	Syntax-directed	14
SD:CODE_SUPERGLOBALS	php.analysis.code.SuperGlobals	Syntax-directed	17
SD:PROT_MYSQL	php.analysis.protocols.MySQLAnalysis	Syntax-directed	9
FP:TYPE_INFERENCE	php.analysis.typing.TypeInference	Fix-point	Chapter 7

Figure 9.4: Analysis identifiers, their corresponding class name, their definition, and reference to Chapter 8.

implemented, and their corresponding class names and references to Chapter 8 is given in Figure 9.4. An overview of the feedback generated by these analyses is found in Appendix A.

9.4.2 Adding a new analysis

When deciding to add a new analysis, the first task is to determine what the type of the new analysis must be (i.e. regular expressions, syntax-directed or fix-point). Then, a new class should be added extending the root class of the chosen analysis type (see Section 9.2 for details) and the required abstract methods must be filled in. Then, the remaining methods that are required for the analysis (e.g. for pre- and post-processing) can be added. These should include tasks such as transformations on the analysis results, but also the generation of feedback (i.e. warning messages), based on the interpretation of the analysis results.

The next step is making the analysis engine aware of the existence of the new analysis. This is achieved by adding a unique identifier name and the corresponding package and class names to the `analyses.dat` file in the `data` directory. Lastly, when invoking the analysis engine on the new analyses, the class path should contain the location of the new analyses, or an exception will be thrown.

As displayed in Figure 9.4, each of the identifier names has a specific prefix, which is used to identify its analysis type. The `RE:` prefix stands for a regular expressions analysis, the `SD:` prefix for a syntax-directed analysis, and finally the `FP:` prefix stands for a fix-point analysis. When adding a new analysis, it is important to remember these prefixes, because these will also determine if an analysis can be used in a composition (which is only possible for syntax-directed analyses). If none of these prefixes are used, then the analysis may not properly work in our framework at all.

Chapter 10

Experimental validation

In this chapter we test the analyses that we implemented and discuss the test results. We describe the various sets of test programs, and discuss the results (i.e. the total number of warnings generated) and the validity of these numbers. Our tests are organised on kind of analysis, one to a section. A full description of all possible warning messages that are generated by the analyses can be found in Appendix A.

10.1 Test cases

We have tested our analyses on the following sets of programs:

Case 1 A small practical assignment for PHP programmers of an Internet Programming course. This was the first assignment of this course, and no experience with PHP was expected of these students. The assignment of approximately 30 groups have been analysed.

Case 2 A larger practical assignment, which was the final programming exercise from an earlier incarnation of this Internet Programming course. We only consider one submission of this assignment, due to the large number of submissions making use of classes in PHP, which our tool does not support.

Case 3 A larger practical assignment, which was the final assignment of the Internet Programming course of Case 1. We consider multiple submissions here, of approximately six groups, and not all files of these groups could be parsed.

Case 4 A large site for news articles and polls, using a content management system to manipulate the site. This site is actually in use (and still is) for over a year and was written by the author of this thesis.

In Figure 10.1 some additional statistics for each of the test cases is listed.

For each of the analyses, we apply it to each of these four cases, and compare the results. The files that could not be parsed by our tool because they don't meet our restrictions (listed in Section 6.2) – approximately 10% of the total number of files – are left out of the discussion.

In the next sections we discuss the results of our experiments, and discuss the kinds of warnings that have been found, illustrated by code examples from our test cases. We also discuss some examples where our analyses may have generated false positives, and explain why these were generated, illustrated by examples.

Case	Files	Lines	Author
Case 1	267	22408	Mainly Bachelor students, but also some master students
Case 2	106	11626	Bachelor students
Case 3	96	16924	Bachelor and Master students
Case 4	85	9346	Author of this thesis

Figure 10.1: Additional statistics for each of the test cases.

Analysis / warning	Case 1	Case 2	Case 3	Case 4
RE:HTML_NONSTANDARD	628	464	157	71
HTML_NON-STANDARD_TAG	330	111	133	48
HTML_NOLOWERCASE	298	353	24	23
RE:HTML_ATTRIBUTES	857	0	431	1296
HTML_REQ_ATTRIB	609	0	243	1294
HTML_ILLEGAL_ATTRIB	248	0	188	2
RE:HTML_ANALYSIS	2658	200	599	8762
HTML_CLOSE_TAG	208	0	106	176
HTML_TAG_NESTING	944	63	205	3154
HTML_UNCLOSED_TAG	1506	137	288	5432
SD:HTML_ANALYSIS	1826	633	530	1884
HTML_CLOSE_TAG	176	0	136	62
HTML_NO_XML_AT_START	157	23	40	59
HTML_TAG_NESTING	544	52	105	1162
HTML_NON-STANDARD_TAG	199	107	42	45
HTML_NOLOWERCASE	257	349	0	23
HTML_UNCLOSED_TAG	493	102	207	532
SD:TAINT_GET_POST	213	110	171	266
CODE_GETPOST	159	107	108	233
CODE_GETPOST_FUNCTION	54	3	63	33

Figure 10.2: HTML analyses test results. The numbers are the amount of warnings for each analysis, and its specific kinds of warnings.

10.2 HTML validation

We have tested these HTML validation analyses:

- `RE:HTML_ANALYSIS`
- `SD:HTML_ANALYSIS`
- `RE:HTML_NONSTANDARD`
- `RE:HTML_ATTRIBUTES`
- `SD:TAINT_GET_POST`

The results of these analyses on our four test cases are displayed in Figure 10.2.

From Figure 10.2 it may look as if the syntax-directed variant of the HTML analysis performs better than the regular expressions variant. It looks like tag nesting is better in the syntax-directed variant, though the program flow is not considered here. Further research is needed if we want to prove that syntax-direct analysis is indeed better than the regular expressions, and that if we also involve program flow (i.e. perform fix-point analysis), that the results of the HTML analysis is even further improved.

The large number of warnings for `HTML_ATTRIBUTES` contain a lot of false positives. This can be explained with the following code example from one of the files being tested:

```
$celOpen = "<td><a href=\"userlist.php?details=" . $poster_id . "\" target=\"_blank\">";
```

Here, the variables are concatenated using the `.` operator. If interpolation was used instead, then the extra `"`'s would have been unnecessary, so that the regular expressions would not have had any problem distinguishing the attributes properly. The kind of warnings that this example gives is:

```
case2/userlist.php:246:
Class: RE:HTML_ATTRIBUTES - HTML_ILLEGAL_ATTRIB
Message: Non-allowed attribute found for tag a
Found: attribute=[ . $poster_id . ]
```

False positives can also be found in the results of the HTML_CLOSE_TAG warning type of the SD:HTML_ANALYSIS. Consider the following example:

```
function printFooterStrict() {
  print "</div> \n";
  print "</body> \n</html> \n";
}
```

Though at first there seems nothing wrong with this example, it only prints closing tags, which are not opened by this function. As an effect, the analysis assumes that these tags are without an opening tag, generating warnings such as:

```
case3/BackOffice/httpheaderfunctions.inc:88:
Class: SD:HTML_ANALYSIS - HTML_CLOSE_TAG
Message: Dangling close tag found: </div>
Found: tag=[</div>]
```

The same analyses performed on the larger practical assignment (Case 2), make it assumable that the HTML quality seems considerably better than that of Case 3, considering the average amount of warnings per file is also lower. The large site (Case 4) however shows a very large amount of warnings for the regular expression analysis for HTML, while the number of warnings for the syntax-directed version is comparable to that of Case 1. This can be explained with the complexity of the site: due to the intensive usage of functions, branching constructs, jumps and terminations with the `die()` function, a lot of extra false positives are generated.

The GET/POST taint analysis finds comparable numbers for each of the cases. This can be explained by the fact that validation of a web variable sometimes takes places in a statement after the web variable has been assigned to a PHP variable. These situations cannot be detected by our analysis:

```
$id = $_GET['post_id'];
$id = str_fix($id);
```

In this example from Case 4, the `str_fix` function performs the untainting. However, the tainted value from the GET variable is first assigned to `$id` directly, so our analysis will assume here that this GET variable does not get properly untainted, which would generate the following warning:

```
case4/news.php:249:
Class: SD:TAINT_GET_POST - CODE_GET_POST
Message: Tainted web input variable: $_GET['post_id']
Found: $_GET=['post_id']
```

The number of warnings for this analysis will most likely be reduced to a neglectable number if all untaint functions are specified and used directly on extracted data. And considering that the number of these is quite low (especially for Case 2), it is safe to guess that in general no mistakes are made in these scripts.

HTML validation by analysis of the source code seems to perform quite well, but is definitely not perfect. Especially the regular expression variant finds a lot of false positives, which the syntax-directed version does not. However, even the syntax-directed version still finds too many problems that are not really there. Also the GET/POST taint analysis should be able to perform better in a fix-point analysis, because that way, fewer false positives will be detected, in cases where input validation takes place after the initial assignment of the web input to a PHP variable (e.g. see the previous example).

It is not certain whether fix-point analysis can improve the HTML analysis much; it would certainly help a lot with finding problems with tag nesting and tag closing, but it will certainly not bring any advantages for the rest of the HTML analyses, because these analyses (e.g. tag attributes and non-standard tags) are performed on the strings itself, and work independently of the program structure.

The values that are within our expectations, are the number of warnings for non-lowercase tags and missing XML declaration header at the start of each script. However, some of the scripts generate `.xml` documents, which are not HTML, and contain tags that are not HTML, possibly with non-lowercase tags. If these specific scripts are left out of our tests, then the number of warnings that are found will be the exact number of wrong tags. An example:

```
case4/rss.php:19:
Class: RE:HTML_NONSTANDARD - HTML_NON-STANDARD_TAG
Message: Non-standard tag found: pubDate
Found: tag=[<pubDate>]
```

Analysis / warning	Case 1	Case 2	Case 3	Case 4
FP:TYPE_INFERENCE	1880	213	190	984
TYPE_CHANGE	208	0	15	40
COERCION_TO_ARRAY	403	0	20	125
MULTI_TYPE_VAR	51	0	11	31
TYPE_CHANGE_ANY	42	3	0	24
FUNCTION_ANY_RESULT	23	0	1	12
FUNCTION_MULTI_TYPE	18	0	0	0
COERCION_BOOL_NUM	4	0	0	0
NEW_VARIABLE	820	173	82	504
TYPE_MULTI_CHANGE	50	19	0	23
COERCION_NUM	133	0	7	103
COERCION_BOOL	78	37	18	2
COERCION_STRING_NUM	68	0	17	120
SD:TYPE_ASSIGNS	71	0	3	139
TYPE_ASSIGN	71	0	3	139
TYPE_INCDEC	0	0	0	0
SD:TYPE_CONDITIONS				
TYPE_CONDITIONS	273	11	67	877
SD:TYPE_CONCAT	12	0	0	384
STRING_IN_NUM	0	0	0	0
NUM_IN_STRING	12	0	0	384

Figure 10.3: Type inference, and type analyses test results. The numbers are the amount of warnings for each analysis, and its specific kinds of warnings.

This tag is generated by a RSS file generator. While it is evident that this is not a valid HTML tag, it definitely is a valid tag for RSS. Our analysis is unable to distinguish between different XML file types (of which XHTML is an example).

10.3 Type inference

For type inference, these analyses have been tested:

- SD:TYPE_CONDITIONS
- SD:TYPE_CONCAT
- SD:TYPE_ASSIGNS
- FP:TYPE_INFERENCE

In Figure 10.3 the test results for these analyses on our test cases are displayed.

Again it appears that the final practical assignments (Cases 2 and 3) generate fewer warnings on average than the first one (Case 1). The large site (Case 4) still generates a lot of warnings, but in total does not really exceed the other cases much.

However, there are some interesting cases that seem rather inaccurate or differ a lot from the other results. Especially the NEW_VARIABLE warning type, which indicates undefined variables or variables of which the initial type cannot be determined. These especially occur when a web input (or session) variable is assigned to a PHP variable, which have the *any* type. Explicit coercions could solve this problem, for example:

```
$num_rows = mysql_num_rows($result);
$num_fields = mysql_num_fields($result);
$row = mysql_fetch_assoc($result);

if ($num_rows) {
```

```
  ...

  for ($i=0; $i<$num_rows; $i++) {
    reset($row);
    ...

    $sprice = querySpecialPrice($row[product_id]);
    if($sprice == -1) {
      ...
    }
    else {
        $row[price] = $sprice;
        print "<tr class=\"specialprice\">";
    }
    print '<td><a href="product.php?id=' . $row[product_id] . '">' . $row[name] . '</a> </td>';
    print makePriceTD($row[price]);

    ...
  }
  ...
```

Because the type of `$row` is *array*[*any*] (which is the standard type that the function `mysql_fetch_assoc` returns), the type of the elements of this array is *any*. When an element of this array would propagate through the program, then these are also assign the *any* type, which leads to a warning such as:

```
case3/FrontOffice/productlistingfunctions.inc:10:
Class: FP:TYPE_INFERENCE - NEW_VARIABLE
Message: Non-concrete type assigned to variable $priceInt
Found: $priceInt=[any]
```

Here, the explicit coercion `(int)$row[product_id]` would solve the problem at the `querySpecialPrice ($row[product_id])` call. Other type problems such as these can be solved similarly.

A similar problem can be found in the warnings for array coercions. These nearly all occur for variables to which the *any* type is assigned (possibly generating a `NEW_VARIABLE` type warning as well), which are actually meant to be used as arrays. This can e.g. happen when an array value is stored in the `$_SESSION` superglobal, which has the default element type *any*.

Another example of coercions that are detected well are the string to numeric coercions, of the `COERCION_STRING_NUM` warning type. For example:

```
$record_posts = mysql_fetch_array($result_posts);
$num_posts = $record_posts['num_posts'];

...

if ($num_posts > 0)
{

...

}
```

In this example the `$num_posts` variable gets assigned the *string* type (because the `$record_posts` variable is of type *array*[*string*], which is the return type of the `mysql_fetch_array` function). But in the `if` statement, it is compared to a number. The warning that is generated by this example is:

```
case1/phpmysql_functions.inc:305:
Class: FP:TYPE_INFERENCE - COERCION_STRING_NUM
Message: string to numerical coercion for variable $num_posts
Expected: $num_posts=[num]
Found: $num_posts=[string]
```

A simple explicit coercion to *int* at the assignment of `$num_posts` would solve this problem.

One example that does not perform well are the FUNCTION_ANY_RESULT type warnings. These are nearly all caused by library functions for which the argument types did not match the ones given by the list of predefined functions (Appendix B.2.2), which leads to this warning as a result. For example:

```
case3/site/footer.php:26:
Class: FP:TYPE_INFERENCE - FUNCTION_ANY_RESULT
Message: Cannot determine a concrete result type for function strcmp with argument types
[[any], [string]]
Found: strcmp=[[[any], [string]] => [any]]
```

By default, the strcmp function expects two string arguments (and compares them), and yields an *int* as result. However, if one the arguments is not of *string* type (but e.g. of *any*), the result type can not be properly determined, thus yielding the *any* type as result.

One way to solve this problem is by adding the missing argument types to the function list (which could lead to large lists of possible arguments, see Appendix B.2.2 for details on adding function types to the function list), or allowing the *any* type to be used for every argument of a function call. However, this could lead to accurate warnings of this type to remain undetected, and definitely makes determining the correct result type for specific argument types impossible.

There are also examples that work very well. For example, type changes of a variable are properly detected, as demonstrated by the following example:

```
$internal = $_POST['internalnew'];

...

if($internal)
  $internal = 1;
else
  $internal = 0;
```

The type of $_POST is *array*[*string*], so its element type is *string*. This leads to $internal getting the *string* type. However, in the if ... else statement, the type of $internal gets changed to *int* by both branches, leading to a TYPE_CHANGE warning after the execution of either branch. If either branch did not change the type (or changed it to even another type, such as *bool*), then a different warning would be given, namely TYPE_MULTI_CHANGE. Another warning thrown by this single example is the COERCION_BOOL warning, because the string $internal is used as a boolean condition in the if ... else statement.

The other warning concerning multi-types (i.e. MULTI_TYPE_VAR) occurs when the type of a variable gets initialized as a multi-type. For example:

```
function getReply($post_id) {
  global $connection;

  $query = sprintf("SELECT post_id FROM posts WHERE reply_id='%d'", $post_id);
  $result = mysql_query($query, $connection);
    if ($result) {
    $post = mysql_fetch_array($result);
    return $post["post_id"];
  } else {
    return -1;
  }
}
```

This function can return both an integer or a string as result, making its result type $\{string, int\}$. This is very problematic when the result of this function gets assigned to a variable, or used as the argument for another function call. This leads to warning messages such as:

```
case1/topicdetail.php:23:
Class: FP:TYPE_INFERENCE - MULTI_TYPE_VAR
Message: Multiple possible types found for variable $post_id
Found: $post_id=[num, string]
```

Analysis / warning	Case 1	Case 2	Case 3	Case 4
SD:PROT_MYSQL	793	0	230	87
MYSQL_ILLEGAL_CALL	730	0	222	32
MYSQL_MISSING_CLOSE	63	0	8	55

Figure 10.4: Test results for the MySQL protocol analysis. The numbers are the amount of warnings for each analysis, and its specific kinds of warnings.

Finding out the exact cause of messages such as these that are caused by this example can be very tricky, especially if it concerns the function parameters having the multi-types. Therefore, programmers should take extra care with determining the result types of their functions. Moreover, the `FUNCTION_MULTI_TYPE` warning that this example also throws, is a good indicator of problems.

In general, the type inference algorithm of Chapter 7 performs very well. Still, there is enough space for improvement, especially on the detection of *any* type assignments to variables. Often these come from web input variables or session variables, and if more information could be provided here, the analysis could become more accurate. Another issue for improvement could be the handling of library functions, especially if for their arguments no appropriate result type can be found. The amount of false positives of our algorithm seems rather low (and falls within our expectations). Further research is required to determine which of the warning types generates the most false positives, and how the quality for these warning types can be improved.

For the syntax-directed analyses, the analysis that checks for assignments in expressions (`SD:TYPE_ASSIGNS`) performs very well. Apparently the prefix/postfix operator is nowhere used in any of the tested scripts. However, assignments in expressions (especially in conditions) appear to be used very often, especially where databases are used a lot (test cases 1 and 4). For example:

```
$result = mysql_query("SELECT * FROM News WHERE ID='$news_id');

while ($row = mysql_fetch_row($result)) {
  ...
}
```

The return type of `mysql_fetch_row` is $\{array[string], bool\}$, so the type of `$row` is possibly not *bool* all the time. Most often this would not give any problems (because all array types are translated to the `true` value in boolean coercions), but using a non-boolean expression as a condition is not clean programming. This code fragment could be transformed into a `for` loop using the `mysql_num_rows` function to iterate over all rows in the database query, however this could be tedious work, especially if the above construction is used very often. The warning message generated for this example would be:

```
case4/news.php:318:
Class: SD:TYPE_ASSIGNS - TYPE_ASSIGN
Message: Assignment expression found
Found: exp=[$row = mysql_fetch_row($result)]
```

The last analysis, the `SD:TYPE_CONCAT` analysis, did not detect any use of strings as argument for the `+` operator. However, numbers are often used in string concatenations, which is actually perfectly fine[1] because number to string coercions never fail.

10.4 Protocols

For protocols, we have only implemented one analysis, the `SD:PROT_MYSQL` analysis. In Figure 10.4 the test results for this analysis on our test cases are displayed.

Again a lot of warnings are generated for Case 1, and much less (or none at all) for the others. Especially the `MYSQL_ILLEGAL_CALL` warning appears much less in Case 3, which means that the order in which MySQL functions must be used, is well-respected in these scripts. The other warning type, `MYSQL_MISSING_CLOSE`,

[1] Though it does not look very professional. The alternative, interpolation, would be a better choice here.

indicates that many scripts miss the `mysql_close()` call at the end. This is not really problematic (because the interpreter closes all open connections automatically), but if such a script is used as an include file in another script, this could give potential problems.

The MySQL analysis performs better than we expected. Even though fix-point analysis would be required to accurately determine the order in which the function calls occur, this analysis already seems to detect most problems accurately. There are however still a lot of possible false positives. For example:

```
<?
$conn = mysql_connect( ... );
mysql_select_db( ... );

...

$result = mysql_query( ... );
while ($row = mysql_fetch_row($result)) { ... }

...

$result = mysql_query( ... );
if (mysql_num_rows($result) == 0) {
  mysql_close();
  die( ... );
}

mysql_query( ... );
mysql_close();
?>
```

In this example there are two points at which the database connection is closed: in the `if` statement's body (which also terminates the script due to the `die()` call), and at the end of the program. Normally this would not be a problem, but our analysis does not consider the flow of the program, which means that both the `mysql_close()` calls are considered, with the `mysql_query()` call in between. This would generate two false positives: one for the last `mysql_query()` call, and one for the `mysql_close()` call. Also, different database connection variables are not tracked, and neither are different query variables. A fix-point analysis could solve both problems, and also add more details for result and query variables.

The warnings generated by this example are:

```
case4/user.php:468:
Class: SD:PROT_MYSQL - MYSQL_ILLEGAL_CALL
Message: This function call might not be respecting the protocol rules: mysql_query
Expected: mysql=[connect, error, errno]
Found: mysql=[query]

case4/news.php:469:
Class: SD:PROT_MYSQL - MYSQL_ILLEGAL_CALL
Message: This function call might not be respecting the protocol rules: mysql_close
Expected: mysql=[connect, error, errno]
Found: mysql=[close]
```

10.5 Coding styles

For coding styles we tested these analyses:

- `SD:CODE_SUPERGLOBALS`
- `SD:CODE_BLOCKSIZE`
- `SD:CODE_GLOBAL_NAMES`

In Figure 10.5 the test results for these analyses on our test cases are displayed.

Analysis / warning	Case 1	Case 2	Case 3	Case 4
SD:CODE_SUPERGLOBALS				
DEPRECATED_SUPERGLOBALS	9	7	6	0
SD:CODE_BLOCKSIZE				
BLOCKSIZE	67	8	5	86
RE:CODE_GLOBAL_NAMES	353	0	125	1815
GLOBNAMES_GLOBAL_UNDECLARED	20	0	3	0
LOCAL_NAME_CLASH_WITH_MAIN	333	0	122	1815
STATIC_NAME_CLASH_WITH_MAIN	0	0	0	0

Figure 10.5: Test results for the coding practices analyses. The numbers are the amount of warnings for each analysis, and its specific kinds of warnings.

These three analyses give a good indication of the programming styles of the authors of each test case. In many cases common variable names are used on the global scope and locally in some of the functions, which is not really a problem, unless these local variables are changed to global variables, in which case they possibly have to be renamed in these functions to avoid name clashes or even loss of data. For example:

```
function foo($v) {

  if ($v == 2) $a = 4; else $a = 3;
  $b = 4;

  return $a * $b;
}

$a = 2;
$d = foo($a);
echo $a * $d;
```

If the `$a` variable gets globally declared at the entry of function `foo`, then the value of `$a` on the global scope becomes overwritten with the value `4` by the `$a = 4` statement in the `foo` function, and this alters the text printed by the `echo` statement in the main program to `64` instead of `32` when `$a` was not declared as global variable. This causes a `LOCAL_NAME_CLASH_WITH_MAIN` warning message in the `RE:CODE_GLOBAL_NAMES` analysis:

```
Class: RE:CODE_GLOBAL_NAMES - LOCAL_NAME_CLASH_WITH_MAIN
Message: Local variable $a in function foo is also declared in the main program
Found: local=[$a]
```

The reason that the scripts of Case 4 have no problems at all with global variables, is because it does not make any use of global variables at all. Because only one database connection is used, no global database variable (e.g. `$db`) has to be declared for each function. It does however use many common variable names in functions (e.g. `$row` and `$result`), which are used for database queries in both the main program and in many of the functions. For Case 2 the analysis does not generate any warning at all, because in these scripts all variable names are completely different from those on the global scope, which means that no name conflicts can occur.

The other warning type for the `RE:CODE_GLOBAL_NAMES` analysis, `GLOBNAMES_GLOBAL_UNDECLARED`, is caused by the fact that some global variables are declared somewhere on the global scope, and are used as some kind of ”shared variable” between functions. Though this often would not cause problems, it can be dangerous. For example:

```
function foo($id) {
  global $db;

  $result = mysql_query("SELECT * FROM USERS WHERE ID = '$id',$db);
  ...
```

```
}

foo(3);
...
```

If the `$db` variable in the above example is not initialized first, then the query will fail, because the `$db` is not a database connection resource, but an empty variable.

Though the `SD:CODE_BLOCKSIZE` analysis may seem quite useless, it does give good information on the complexity of the scripts themselves. If more large code blocks are used, then the script seems to be more complex (or it is simply just larger). Also, large code blocks easily get complicated to understand, and harder to maintain, so it can be used as a good indicator of possible future maintenance issues.

10.6 Conclusion

The general picture displayed by our test results, is that more experienced PHP programmers seem to get fewer warnings than less experienced ones. In Cases 2 and 3, fewer warnings than in the first test case have been generated, because these scripts are generally of much better quality than those of Case 1. It is also evident that Case 4 is more complex of structure and size than any of the other cases, and this is well-reflected by the results of the coding style analyses. Because Case 4 is one large site, and not several (smaller) ones, it can be expected that more warnings are generated for the HTML validation and coding styles, because these analyses are typically more dependent on the complexity of a site.

Most of our analyses appear to work very well, and it seems evident that the regular expression analyses perform worse than their syntax-directed variants. However, more research will be required to prove if syntax-directed analyses are superior to ones using regular expressions. And if we also implement fix-point based versions of the same analyses, we expect that it is likely to show even more accuracy and fewer false positives. Again, more research is required to prove this statement. Still, we feel that our analyses do give quite accurate results, and behave exactly as we had expected, for each of the test cases.

Chapter 11

Conclusion

We have designed and implemented a system that performs static analyses on PHP, namely type inference, HTML validation and several coding style related analyses. In this chapter we discuss some work related to our own, and conclude with a list of future work, and expectations.

11.1 Related work

There have been several projects on the subject of static analysis on PHP and related languages, of which most are focused on detection of vulnerabilities in source code, that could be used to perform attacks such as cross-site scripting (XSS), and SQL injection. We focused mainly on correctness and validation, so there are quite a few differences between our work and these other projects. In this section we discuss a few of such projects which perform analysis on PHP.

The first to deal with static detection of web application vulnerabilities in PHP were Huang et al. [29], who used a lattice-based analysis algorithm derived from type systems [46] and typestate [43], and compared it to a technique based on bounded model checking in their follow-up paper [28]. Their system, WebSSARI, had promising results, but unfortunately was unable to deal with several technical issues. For example, a substantial fraction of the PHP files they tested (approximately 8%) was rejected due to problems with the applied parser. Furthermore, the handling of object references and array accessing was not mentioned, and files are not included automatically. Also, WebSSARI was focused on detection of vulnerabilities, not on correctness of PHP code.

Pixy [33], a project by Jovanovic et al., is a Java-based tool that uses a combination of several analyses to detect code vulnerabilities. It is based on a context-sensitive, inter-procedural data-flow analysis that, combined with *alias analysis*, is able to deal with references and accessing arrays, to detect taint-style vulnerabilities. The goal of this analysis system is to determine whether it is possible that tainted data reaches a sensitive sink – i.e. a place where data leaves the program, e.g. by being printed by the browser, or e.g. written to a file or database – without being properly sanitized. Pixy does not analyse PHP directly, but operates on the control-flow graph of a program. In contrast to Huang et al. [29, 28], Pixy deals with file inclusion automatically. There are still some technical issues, which concern the fact that file inclusions are dynamic, which means that the filename to be included can be constructed at run-time. Another issue is the fact that Pixy does not recognize differences between vulnerabilities, because it uses a safe/unsafe state for specific type of vulnerabilities.

Both the two systems described above do not make use of object-oriented features of PHP. Though Pixy is able to parse the full PHP language (while WebSSARI is unable to), in a later version of this tool it may be able to deal with these constructs. However, new PHP code will probably make much use of the object-oriented paradigm.

The third tool we discuss here, PHP-Sat [4] by Bouwers, is able to deal with the full PHP language. Aside of being able to deal with security vulnerabilities like WebSSARI and Pixy do, this tool can also look for several bug patterns that are potential problems, much as our own analyses are capable of doing as well. PHP-Sat uses a security algorithm based on typestate [43] and the work of Huang et al. [29, 28], and uses a set of safety types, Γ, which follows the properties of the lattice model of secure information flow by Denning [12].

This model is more advanced than that of Pixy, because it is able to distinguish more types of security levels, so that more sensitive safety information of variables is possible. These safety types are assigned to variables, function operations and the sensitive sinks, so that detection of vulnerabilities becomes basically checking if a sensitive sink does not receive values that do not meet the sensitive sink's safety requirements. Tracking down the source of the security vulnerability then becomes quite straightforward.

The last project that we discuss is that of [38], which uses a technique for approximating the string output of PHP programs. This project is primarily targeted at HTML validation by analysing the source code as we have done, but the author also claims that his work can be used for detection of cross-site scripting vulnerabilities. However, it is unable to distinguish between malicious and benign output, because no taint information or any checks that gather this kind of information are performed by the tool. However, the idea of transforming a PHP program to a context-free grammar that can be used to perform HTML validation with, seems quite promising.

11.2 Future work

We see several ideas for future work. Firstly, our tool cannot handle object-oriented features of PHP, and several other constructs as well. We had omitted the implementation for these, so that we could focus on the core features of PHP, and the most used features, so that our tool would run for as many PHP programs as possible. However, in order for our tool to become useful for commercial applications, we would need to have a mature parser and analyser that can handle all of PHP and its features.

In Section 7.6 we discussed some ideas that could be used to extend our type inference system. This not only includes the support for classes, but also for the full functionality of global and static variables. In our approach we made the assumption that global variable declarations appear at the top of a function, followed by static function declarations and the actual body of the function. However, PHP allows these declarations everywhere inside of a function, even allowing alternating between both scopes. This could be implemented by adding another type of *context* on these variables, aside of the function call context that we already have. This way, we would be able to distinguish between either of the three scopes (local, global and static), and switch between them accordingly, wherever needed.

We have not implemented all analyses that we have specified in Chapter 8, nor have we even specified all possible analyses that could be thought of using the theories that we described in Chapters 4 and 5. Especially the taint analysis would be a welcome addition to our system. For this analysis we propose the same approach that we use in our type inference system, i.e. constraints, combined with typestates [43] to distinguish the safety level of variables. These constraints could make use of the two kinds that we described in our system: normal constraints to assign safety types to variables upon assignment or their creation in the PHP program, and expected constraints, that are attached to sensitive sinks that demand their arguments to be of a minimum safety type. Using the same approach as with our coercion detection in the type inference system, the expected safety types and the found safety types could be matched to each other, resulting in generation of warnings where the expected constraints are violated.

Chapter 10 discusses the occurrences of false positives in several of our analyses. Most often, these are generated due to the fact that these analyses are written on a lower accuracy level than intended. For example, the MySQL protocol analysis works best as a fix-point analysis on the control-flow of a program, instead of a syntax-directed analysis on the AST. Even though the syntax-directed analysis already gives quite promising results, it cannot cope with the subtleties of the program flow of a PHP program, and neither handle recursion properly. Another problem with the syntax-directed analyses is that these cannot handle the propagation of the variables that are involved in the protocols properly. For example, a PHP program could contain multiple database connections, using multiple query resources. Our analysis cannot detect this, and therefore could generate false warning messages when one such connection is closed. We expect that a fix-point analysis solves this problem and others.

Another issue with accuracy of analyses is the difference between regular expressions, and syntax-directed analyses. We only implemented the HTML analysis in both styles, of which the regular expressions variant showed significantly more false positives than the syntax-directed variant. Clearly it seems that syntax-directed analyses are more accurate, though more (empirical) research is required to prove this claim. Furthermore, we believe that fix-point analysis is even more accurate, yet this still has to be proved also, though it not always sure what the benefits of such an analysis would be.

Lastly, we do not share information between analyses. In our implementation this was not necessary, because each analysis was designed to work independently. However, some analyses could be developed, that could make use of the results that are calculated by other analyses. This however, would add dependencies between analyses, which would mean that certain analyses cannot be performed, unless these dependencies are met. It would also add a whole range of new problems, such as cyclic dependencies of analyses requiring the results of each other.

Bibliography

[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis Symposium*, pages 78–100, 1994.

[2] A. Aiken and L. Wimmers. Soft typing with conditional types. In *Proc. of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[3] J. Aycock. Aggressive type inference. In *The Eighth International Python Conference*, 2000-01.

[4] E. Bouwers. Analyzing PHP: An introduction to PHP-Sat. Center for Software Technology, University of Utrecht, Netherlands, Jan. 2007.

[5] T. Bunce. DBI - Database independant interface for Perl. Available at `http://search.cpan.org/~timb/DBI-1.51/DBI.pm`.

[6] B. Cannon. Localized type inference of atomic types in Python. Master's thesis, Faculty of the California Polytechnic State University, June 2005.

[7] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292. ACM Press, NY, USA, 1991.

[8] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically typed object-oriented programs. In *LISP AND SYMBOLIC COMPUTATION: An International Journal*. Kluwer Academic Publishers, 1991.

[9] World Wide Web Consortium. The W3C Markup Validation Service. `http://validator.w3.org/`.

[10] World Wide Web Consortium. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). W3C Recommendation 26 January 2000, revised 1 August 2002, available at `http://www.w3.org/TR/xhtml1/`, 2002.

[11] L. Damas and R. Milner. Principle type schemes for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 278–292, 1982.

[12] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[13] Perl Documentation. Perl regular expressions. `http://www.perl.com/doc/manual/html/pod/perlre.html`, 2001.

[14] C. Esterbrook. Introduction to Webware for Python. In *9th International Python Conference*, 2001-02.

[15] C. Esterbrook et al., J. Love, G. Talvola, and I. Bicking. WebKit User's Guide. Available at `http://www.webwareforpython.org/WebKit/Docs/UsersGuide.html`, 23 April 2006.

[16] C. Hanson et al. *MIT Scheme Reference Manual*. MIT, 1.96 for Scheme release 7.7.0 edition, March 2002.

[17] J. Rees et al., W. Clinger. Revised report on the algorithmic language Scheme. *ACM Lisp Pointers IV (July-September 1991)*, 1991.

[18] Lars T. Hansen et.al. ECMAScript 4 type system. Available at `http://developer.mozilla.org/es4/clarification/type_system.html`, June 2006.

[19] D. Flanagan. *JavaScript: The Definitive Guide, Fourth Edition*. O'Reilly, November 2001.

[20] Python Software Foundation. Python Database API Specification 2.0. Available at `http://www.python.org/dev/peps/pep-0249/`, April 1999.

[21] J. Goyvaerts. *Regular Expressions - The Complete Tutorial*. February 2006.

[22] ECMAScript 4 group. ECMAScript 4 proposals. Available at `http://developer.mozilla.org/es4/proposals/proposals.html`, June 2006.

[23] The PHP Group. PEAR :: The PHP Extension and Application Repository. `http://pear.php.net`.

[24] The PHP Group. PHP: Hypertext Processor. `http://www.php.net`.

[25] The PHP Group. PHP: PHP Usage Stats. `http://www.php.net/usage.php`.

[26] The PHP Group. PHP: Strings - Manual. `http://www.php.net/types.string`.

[27] J. Hildebrand. Automatic HTML Validation. Available at `http://wiki.webwareforpython.org/automatic-html-validation.html`, 20 Sept 2002.

[28] Y. Huang, F. Yu, C.Hang, C. Tsai, D. T. Lee, and S. Kuo. Verifying web applications using bounded model checking. *dsn*, 00:199, 2004.

[29] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM Press.

[30] J. Hunter and W. Crawford. *Java Servlet Programming, Second Edition*. O'Reilly, April 2001.

[31] G. Jackson. Securing Perl with type inference. `http://www.umiacs.umd.edu/~bargle/project2.pdf`, May 20, 2005.

[32] E. H. Jansson. TWINE Project. Available at `http://twineproject.sourceforge.net/index.html`.

[33] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities, 2006.

[34] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[35] M. Maischein. `Test::HTML::Content` - Perl extension for testing HTML output. Available at `http://search.cpan.org/dist/Test-HTML-Content/lib/Test/HTML/Content.pm`.

[36] A. Martelli. *Python in a Nutshell*. O'Reilly, March 2003.

[37] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, apr. 1978.

[38] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM Press.

[39] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Programming Analysis*. Springer-Verlag, 2005.

[40] T. S. Norvell. The JavaCC FAQ. Available at `http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq.pdf`, July 6, 2005.

[41] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, New York, NY, 1991. ACM Press.

[42] S. Spainhour, E. Siever, and N. Patwardhan. *Perl in a Nutshell, Second Edition*. O'Reilly, June 2002.

[43] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.

[44] K. Tatroe, R. Lerdorf, and P. MacIntyre. *Programming PHP, Second Edition*. O'Reilly, April 2006.

[45] A. K. Wright. *Practical soft typing*. PhD thesis, Houston, TX, USA, 1996.

[46] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.

Appendices

Appendix A

Analysis details

This appendix lists the details about the warning types for each of the analyses that have been implemented in the PHP Validator tool. All of these are listed in tables, organised by analysis type. Each table contains a short description of the analysis, a reference to its full description in Chapter 8, and the identifier used in our implementation. For the warning types, we give the identifier, a short description, and the priority levels. These priorities are based on our own experience with these warning types, and can be altered to the needs of the tool user.

A.1 HTML validation

- **Identifier** `RE:HTML_ANALYSIS`

  Analysis Missing closing tags, Tag nesting (Analyses 1 and 3).

  Description This analysis checks that for each opening tag, a closing tag exists as well. Additionally, it also checks if tag nesting is done correctly.

Identifier	Priority	Message	Description
`HTML_CLOSE_TAG`	0.6	Dangling close tag found: `<tag>`	Generated when a closing tag is found but no corresponding opening tag can be found.
`HTML_TAG_NESTING`	0.4	Unmatching close tag found: `<tag>`	Generated when a closing tag is found, but it does not match the last opening tag.
`HTML_UNCLOSED_TAG`	0.8	Unclosed tag found: `<tag>`	Generated when no closing tag can be found for this opening tag.

- **Identifier** `RE:HTML_NONSTANDARD`

  Analysis Non-standard tags (Analysis 2).

  Description This analysis checks for every tag that it is a valid tag specified by the W3C standard for XHTML 1.0 Strict.

Identifier	Priority	Message	Description
`HTML_NON-STANDARD_TAG`	0.3	Non-standard tag found: `<tag>`	Generated when a non-standard tag has been encountered.
`HTML_NOLOWERCASE`	0.3	Non-lowercase tag found: `<tag>`	Generated when a non-lowercase tag has been encountered.

- **Identifier** `RE:HTML_ATTRIBUTES`

  Analysis Tag attributes (Analysis 4).

  Description This analysis checks whether tags have illegal attributes or not.

Identifier	Priority	Message	Description
HTML_ILLEGAL_ATTRIB	0.3	Non-allowed attribute found for tag <tag>	Generated when an illegal attributes has been encountered.
HTML_REQ_ATTRIB	0.1	Missing required attribute found for tag <tag>	Generated when a required attribute is missing.

- **Identifier** SD:HTML_ANALYSIS

  Analysis Missing closing tags, Tag nesting (Analyses 1 and 3).

  Description This analysis checks that for each opening tag, a closing tag exists as well. Additionally, it also checks if tag nesting is done correctly.

Identifier	Priority	Message	Description
HTML_CLOSE_TAG	0.6	Dangling close tag found: <tag>	Generated when a closing tag is found but no corresponding opening tag can be found.
HTML_TAG_NESTING	0.4	Unmatching close tag found: <tag>	Generated when a closing tag is found, but it does not match the (last opened) one.
HTML_UNCLOSED_TAG	0.8	Unclosed tag found: <tag>	Generated when no closing tag can be found for this opening tag.
HTML_NO_XML	0.1	Expected: <?xml ... ?> tag	Generated when the required XML declaration tag cannot be found.
HTML_NO_DOCTYPE	0.1	Expected: <!DOCTYPE ... > tag	Generated when the required doctype declaration tag cannot be found.

- **Identifier** SD:TAINT_GET_POST

  Analysis Validation of GET and POST variables (Analysis 5).

  Description This analysis checks if web input variables from GET/POST context are validated before being allowed to propagate through the program.

Identifier	Priority	Message	Description
CODE_GETPOST	0.75	Tainted web input variable: foo	Generated when for a web input variable no validation is performed, before it is assigned to a PHP variable.
CODE_GETPOST_FUNCTION	0.2	Web input variable found, but unsure if 'foo' is an untaint function	Generated if a function is applied to a web variable, but the analysis is unable to determine if this function is an untaint function.

A.2 Type inference

- **Identifier** SD:TYPE_CONDITIONS

  Analysis Expressions of conditional statements (Analysis 6).

  Description This analysis checks that expressions of conditional statements such as `if ... else` and `while (...) { ... }` are of boolean type.

Identifier	Priority	Message	Description
TYPE_CONDITIONS	0.2	Possible non-boolean condition found	Generated when for an expression it cannot be determined if it is of a boolean type.

- **Identifier** SD:TYPE_CONCAT

Analysis String concatenation: consistent usage of the `+` and `.` operators (Analysis 7).

Description This analysis checks that expressions using the `+` operator must have numerical operands, and that expressions using the `.` operator have (preferably) string operands.

Identifier	Priority	Message	Description
`STRING_IN_NUM`	0.4	String expression found in numerical addition expression	Generated when a string expression is encountered in a numerical expression.
`NUM_IN_STRING`	0.0	Numerical expression found in numerical string concatenation	Generated when a numeric expression is encountered in a string concatenation.

- **Identifier** `SD:TYPE_ASSIGNS`

  Analysis Assignments in expressions (Analysis 8).

  Description This analysis checks if side effects exist in expressions, of the form `$a = $b`, `$a++`, or a combination of both.

Identifier	Priority	Message	Description
`TYPE_ASSIGN`	0.2	Assignment expression found	Generated when an assignment expression has been encountered.
`TYPE_INCDEC`	0.3	Prefix/postfix inc/dec expression found	Generated when a prefix or postfix inc or dec expression has been encountered inside another expression.

- **Identifier** `FP:TYPE_INFERENCE`

  Analysis Type inference algorithm (Chapter 7).

  Description Performs full type inference on the script.

Identifier	Priority	Message	Description
`NEW_VARIABLE`	0.8	Undefined variable found: `$a`	Generated when a variable is used for which no type has been determined.
`NEW_VARIABLE`	0.1	Non-concrete type assigned to variable `$a`	Generated when the *any* type is assigned to a newly declared variable.
`COERCION_NUM`	0.0	*int* or *float* to *num* coercion for variable `$a`	Generated when a coercion of *int* or *float* to *num* is encountered.
`COERCION_FLOAT_INT`	0.3	*float* to *int* coercion for variable `$a`	Generated when a coercion of *float* to *int* is encountered.
`COERCION_STRING_NUM`	0.4	*string* to numerical coercion for variable `$a`	Generated when a coercion of *string* to a numerical type is encountered.
`COERCION_BOOL_NUM`	0.1	*bool* to numerical coercion for variable `$a`	Generated when a coercion of *bool* to a numerical type is encountered.
`COERCION_OBJECT`	0.4	*object* to non-*object* coercion for variable `$a`	Generated when a coercion of *object* to any non-*object* type is encountered.
`COERCION_BOOL`	0.3	*bool* to non-*bool* coercion for variable `$a`	Generated when a coercion of *bool* to any non-*bool* type is encountered.
`COERCION_TO_ARRAY`	0.4	non-*array* to array coercion for variable `$a`	Generated when a coercion of a non-*array* to *array* type is encountered.
`TYPE_CHANGE_ANY`	0.3	Type has changed to the ambiguous type [*any*] for variable `$a`	Generated when the *any* type is assigned to a variable.
`TYPE_CHANGE`	0.5	Type has changed for variable `$a`	Generated when the type of a variable changes.
`MULTI_TYPE_VAR`	0.7	Multiple possible types found for variable `$a`	Generated when a variable gets assigned a union type.

Identifier	Priority	Message	Description
TYPE_CHANGE_VOID	0.5	Void result type assigned to variable $a	Generated when a variable gets assigned the empty *void* type.
FUNCTION_MULTI_TYPE	0.5	Function foo has multiple result types	Generated when a function is found that has more than one result type.
FUNCTION_UNDEFINED	0.9	Undefined function foo found, with argument types [*int*, *bool*]	Generated when a function is found for which no definition is given for the specified argument types.
FUNCTION_ANY_RESULT	0.75	Cannot determine a concrete result type for function foo with argument types [*int*, *bool*]	Generated when a library function is found for which no concrete result type can be determined.
TYPE_MULTI_CHANGE	0.7	Type has changed to a multi-type for variable $a	Generated when the type of a variable changes from a single type to a multi-type.

A.3 Protocols

- **Identifier** SD:PROT_MYSQL

  Analysis MySQL protocol analysis (Analysis 9)

  Description This analysis checks if the MySQL database handling protocol is respected.

Identifier	Priority	Message	Description
MYSQL_UNKNOWN	0.0	Unknown function call: mysql_XXX	Generated when a function for this protocol is encountered which is not in our function list. This can be solved by simply adding the behaviour of this function to the list.
MYSQL_ILLEGAL_CALL	0.3	This function call might not be respecting the protocol rules: mysql_XXX	Generated when a function is encountered that may not respect the order in which the MySQL functions are supposed to be called.
MYSQL_MISSING_CLOSE	0.5	The mysql_close operation is missing	Generated when the MySQL protocol is not properly closed.

A.4 Coding styles

- **Identifier** SD:CODE_BLOCKSIZE

  Analysis Code blocks (Analysis 14)

  Description This analysis checks that no code block exceeds a given maximum amount of statements.

Identifier	Priority	Message	Description
BLOCKSIZE	0.1	Code block too large	Generated when a code block has been encountered that is considered to be too large.

- **Identifier** RE:CODE_GLOBAL_NAMES

  Analysis Names of global variables (Analysis 18)

  Description This analysis checks if variable names that are used as global variables in some functions, are used as non-global variables in other functions.

Identifier	Priority	Message	Description
LOCAL_NAME_CLASH_WITH_MAIN	0.4	Local variable $a in function foo is also used in the main program	Generated when a local variable which is not global, is also found in the main program.
GLOBNAMES_GLOBAL_UNDECLARED	0.15	Globally declared variable $a in function foo not used in main program	Generated when a globally declared variable is not used first by the main program.
STATIC_NAME_CLASH_WITH_MAIN	0.3	Possible name clash for statically declared variable $a in function foo with main program	Generated when a statically declared variable also exists on the global scope.

- **Identifier** SD:CODE_SUPERGLOBALS

  Analysis Deprecation of identifiers (Analysis 17)

  Description This analysis checks if deprecated names of library functions, variables and constants are used.

Identifier	Priority	Message	Description
DEPRECATED_SUPERGLOBALS	0.1	Deprecated superglobal: $VAR	Generated when a deprecated superglobal is encountered.

Appendix B

Installation and configuration

This appendix provides some information about installation of the prototype tool, and how to configure it. The prototype has several configuration files that the user can edit to modify the behaviour of several analyses, and to perform analyses, and suppress certain kinds of warnings.

B.1 Download and installation

The implementation can be found by doing a checkout on the following SVN repository:

```
svn checkout https://svn.cs.uu.nl:12443/repos/php-validator
```

Inside, a JAR file and the source code is found, with all of the required configuration files in the `data` directory. The JAR file is started with:

```
java -Xmx256m -jar php-validator.jar
```

B.2 Configuration files

In this section we discuss the several configuration files that PHP Validator uses. In all of our configuration files, comments are started by the `#` character, and these must appear at the beginning of each line (and may not be preceded by whitespace).

B.2.1 Settings configuration

The PHP Validator tool is preferably invoked using a settings file, which specifies which kinds of analyses have to be performed, and which messages should not be shown in the tool's results. It also specifies some options that cannot be set otherwise, when using the `analyseFile` method of the `AnalysisEngine` class.

The syntax of this file is quite straightforward, as shown in the example below:

```
#######################################################
##### Sample config file for PHP Validator
#######################################################

source=case1
addinclude=1
untaintid=untaint_
blocksize=30

analysis=FP:TYPE_INFERENCE
context_bound=0
suppress=COERCION_STRING_NUM for $id
```

```
analysis=SD:CODE_GLOBAL_NAMES
```

The first four options, have to appear before any specification of analyses. The `source` option specifies the file name (or directory) on which all analyses must take place. The `addinclude` option indicates that include files must be fully expanded for each analysis. If this is the case, then the value `1` must be chosen, otherwise the value `0` should be used. The option `untaintid` specified the prefix or function name that the untaint functions for the `SD:TAINT_GET_POST` analysis should check for. Lastly, the option `blocksize` determines the total number of statements that decide whether a code block in the `SD:CODE_BLOCKSIZE` analysis is considered to be too large.

The analyses are specified by the `analysis` option. Only the analyses that are specified by this option are executed. The `suppress` option can only appear under an `analysis` option, and suppresses one or more warning types for this specific analysis. It is also to specify partial suppression for such a warning type. To achieve this, the `for` keyword must be used directly after the warning type identifier, and then the variable (or tag, depending on the kind of analysis) can be specified.

The type inference analysis (with identifier `FP:TYPE_INFERENCE`) has two additional options: `context_bound`, which is used to specify a maximum bound on its call strings, and `max_resultset`, which defines the maximum number of types that can be in a type set.

B.2.2 Other configuration files

The PHP Validator tool can be configured with many different files, affecting different kinds of analyses. In this section we describe the syntax of these files, and how the user of the tool could extend these, or add his own information to these files. Most of these files are currently incomplete, so it is very welcome to complete them!

Function List

In the `data` directory of the tool a file named `function-types.types` can be found, which contains the types for over 500 of the most commonly used functions in PHP. Adding extra function signatures is pretty straightforward. The format of the file is as follows:

$$\texttt{foo=[[[type}_1^1\texttt{,...,type}_n^1\texttt{],[type}_1^2\texttt{,...,type}_n^2\texttt{],...,[type}_1^m\texttt{,...,type}_n^m\texttt{] => [restype}_1\texttt{,....,restype}_k\texttt{]],}$$
$$\texttt{[[type}_1^1\texttt{,...,type}_n^1\texttt{],[type}_1^2\texttt{,...,type}_n^2\texttt{],...,[type}_1^m\texttt{,...,type}_n^m\texttt{] => [restype}_1\texttt{,....,restype}_k\texttt{]]]}$$

The types are comma-separated. Each function type is a list of arguments (surrounded by brackets), followed by an arrow, and finally a set of result types. All arguments and result types must be sets of base types such as *int* or *array*[*string*] (and may not contain any function types). For example:

```
mysql_query=[[[string]] => [array[string]]]
```

It is also possible to use type variables of the form `t1 ... tn` to use as alias for any member in a set of types, or for polymorphic functions. This also slightly changes the syntax of a function definition:

```
substr_replace=[[[t1], [string], [int]] => [t1], [[t1], [string], [int],[int]] => [t1]]
  where t1 in [string, array[string]]
array_fill=[[[int], [int], [t1]] => [array[t1]]]
```

The tool itself will then try to fit function calls into the smallest possible instance of such a 'polymorphic' type. Lastly, constants and superglobals can be defined likewise, but using a normal typeset instead of a function typeset. For example:

```
$_HTTP_SESSION_VARS=[array[any]]
E_ALL=[int]
```

HTML Validation

For the HTML validation analyses there are three separate config files found in the `data` directory. The `html-tags.txt` file specifies all valid HTML tags, and which tags look structurally the same. This structural equality can be used to reduce the number of false positives for the `RE:HTML_ANALYSIS` analysis, especially for the `HTML_CLOSE_TAG` warning type. An example:

```
td=th
th=td
```

This example declares the `<td>` and `<th>` tags to be structurally equal, so that if one occurrence of `<td>` is closed by a `</th>` tag (or vice versa), it will not make the analysis demand the `</td>` (or `</th>` if `<th>` opening tag was used) anymore.

The `html-singleton.txt` file is used to specify a list of tags that do not have a closing tag. This file contains one tag per line, and is basically just a list of these. The last file, `html-attributes.txt`, specifies the allowed and required attributes for each tag. The syntax is as follows:

```
tag=attrib1,attrib2,...,attribn req_attrib1,req_attrib2,...,req_attribn
```

First the tag is specified, then after the `=` character first a list of allowed attributes is given, optionally followed by a semicolon and a list of required attributes. Both lists are comma-separated. An example:

```
a=class,href,target,onMouseOver,style;href
```

This specifies that the `<a>` tag may have the `class`, `href`, `target`, `onMouseOver` and `style` attributes, but only the `href` attributes is specified to be required for every instance of the `<a>` tag.