Soft Typing PHP

Patrick Camphuijsen

Jurriaan Hage

Stefan Holdermans

Technical Report UU-CS-2009-004
February 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

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

Soft Typing PHP with PHP-validator

Patrick Camphuijsen Jurriaan Hage Stefan Holdermans

Department of Information and Computing Sciences, Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
patrick.camphuijsen@gmail.com jur@cs.uu.nl stefan@cs.uu.nl

Abstract

PHP is a popular language for building websites, but also notoriously lax in that almost every value can be coerced into a value of any imaginable type. Therefore it often happens that PHP code does not behave as expected.

We have devised a flexible system that can assist a programmer in discovering suspicious pieces of PHP code, accompanied by a measure of suspicion.

The analysis we employ is constraint-based, uses a limited amount of context to improve precision for non-global variables, and applies widening to ensure termination.

We have applied the system to a number of implementations made by programmers of various degrees of proficiency, showing that even with these technically rather simple means it is quite possible to obtain good results.

1. Introduction

PHP is a dynamic language which means that variables can be introduced at will, without any explicit declaration, and that any particular variable can be assigned values of many different types throughout its lifetime.

Although types are not visible in PHP code, they are still part of the programmer's conceptual universe. An operator such as addition does expect its arguments to have a type that allows addition, like `int` or `float`. If such is not the case, then values are silently coerced into values of a type that is compatible with the operation in question. In fact, for (almost) any pair of types, such a coercion exists. However, some coercions make more sense than others. For example, a value `37` of type `int` can (losslessly) be coerced to the string `"37"`, but all array values are coerced to the string `"Array"`. The loss of information in the second coercion is quite large, and we suspect that such a coercion was not really what the programmer intended.

Detecting such suspicious coercions is not something that the PHP interpreter is suited for. Therefore, it would be useful to have a tool that helps (novice) PHP programmers to obtain a list of such suspicious coercions (and many other potential problems besides). In this paper we discuss such a tool, PHP-validator.

In a strong typing situation, all errors need to be resolved before a program can be run. Typically the programmer considers only the first error, because usually one can depend on that being a real mistake, and not simply the consequence of another mistake somewhere in the program. Indeed, many compilers only provide a single type error to at the time.

For our system, the situation is quite different. First of all, not every warning signals a mistake (false negatives exist), and second, some mistakes are more serious than others. Therefore, the tool lists quite a number of potential mistakes at once, and the task of the programmer is then to find those that indicate the presence of a bug. To lighten this task as much as possible, two facilities have been built into our tool: a suppression mechanism that allows the programmer to selectively suppress warnings (to help deal with false negatives), and a prioritization mechanism that allows the programmer to control the severity of each particular kind of mistake.

The remainder of this paper is structured as follows. In Section 2 we discuss the PHP language, in particular, those parts that influence the design of the system and some of the assumptions we make about the programs we analyse. We then proceed in Section 4 with a description of the (constraint-based) type system we employ. In Section 5 we describe the fixed point algorithm to compute solutions to the constraints given by the type system, which is made terminating by employing a widening function. Section 6 gives some more details about the tool itself. The results of applying our implementation to a number of PHP applications written by students at various moments during their computer science curriculum can be found in Section 7. In Section 8 we consider related work and Section 9 concludes.

2. The PHP language

In this section we shortly discuss the PHP language, focusing in particular on those aspects that play a role in the type system discussed in the remainder of this paper.

PHP 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". Soon the language became popular among web programmers, as an alternative to systems and languages such as ASP.NET and Java Server Pages, that work in a similar fashion.

The language has a strong webpresence. We mention in particular the excellent website [10] that serves both as a language reference as well as a forum for discussing the ins and outs of the language.

Although, there is no room to fully discuss all the features of the language, we shall highlight some of the more interesting aspects below. Here, we sometimes sacrifice precision to increase concision. The complete syntax specification of the part of PHP that we support can be found in Figure 6.1 of [4].

The PHP version we consider here is PHP 5.0. The language supports many of the usual types: boolean, integers, floating point numbers, strings, arrays and objects, but also resources which can

ISSTA'09,

be understood as references to external entities such as files. Below we shall often refer to the type num, which represents the union of integers and floating point numbers. Indeed, many arithmetical operators are defined to take nums and deliver num. Finally, some functions may be passed values of function type, so called callback functions.

Although the language does have a notion of types, the interpreter will never complain when you use a value of a particular type in a context where another type is expected[1]. For example, the addition operator expects two values of type num. If you pass it, say, a string, the value is coerced into a numerical value according to the rules specified by the language. In this particular case, the string is parsed to see if it starts with something that seems like a number; if the string does not start with a number, then the coerced value is either `0` or `0.0`, depending on the exact context. For example, the string `"12 monkeys"` is coerced to `12` in a context where an integer is expected. For every pair of types, the language definition specifies how the transformation should be performed. Some of these coercions introduce a large loss of precision. For example, any value of type array coerces to the same string value `"Array"`. Although coercions occur silently, explicit type casts are also supported. As we shall later clarify, this distinction is important within the context of our system.

PHP is a dynamic language in the sense that variables can be introduced at will, simply by introducing them in a statement or expression. If a variable is first referred to in a non-assigning context, then the type of that context determines the type and through that type the default value for the variable, e.g., the empty string in case of strings, and `0.0` in the case of floating point numbers. Again the language specifies exactly for each type what those values are. If a variable is assigned to as it first occurs, its value, and therefore also its type, are determined by the assigned value. It is typical for dynamic languages that the type of a variable need not be fixed: a later assignment to the same identifier may change its type.

PHP is in many ways like Perl, but many of the obscure features of that language have been omitted or changed. For example, Perl is a language that supports both static and dynamic scoping for local variables. However, PHP only allows static scoping, and is similar in that to most other languages of its kind.

Another distinction from Perl is that variables that are used or assigned to in a function are, unless specified otherwise, considered to be local to that function. In Perl such variables are, unless specified otherwise, global. In PHP the keywords `global` and `static` can be used to change to the scope of a particular variable. Indeed, PHP even allows the programmer to switch between scopes and access different values associated with variables of the same name but residing in different scopes. The reader may want to try out the code in Figure 1 in his favourite PHP interpreter.

The PHP language introduces a substantial number of built-in variables, often for communicating with the outside world. These variables may contain environment settings, `$_ENV`, or values passed to the script from an HTML form, `$_POST` and `$_GET`. All of these variables exist at the global level, so to refer to any of these inside a function, the programmer would need to explicitly declare them global every single time. This is avoided by allowing to declare variables to be *superglobals*, making their scope global by default when used inside a function. A programmer can define his own superglobals, but more important are those provided by PHP itself, the *autoglobals*. Example include `$GLOBALS` that contains all the global variables (which offers a way to access the global variant of an identifier name without losing the local version), and `$_ENV, $_POST`, and `$_GET`. Note that in previous versions of PHP these variables had different names, like `$_HTTP_GET_VARS`. For legacy reasons, we support both forms, although the latter form is currently deprecated.

[1] Except when you use a non-resource value where a resource is expected.

```
<?php
function staticgloballocalmess()
{
    $bad = 123;
    $badref =& $bad;
    echo "$bad\n";
    global $bad;
    echo "$bad␣and␣$badref\n";
    $bad = 345;
    static $bad;
    echo "$bad␣and␣$badref\n";
    $bad = 919;
    echo "$bad␣and␣$badref\n";
    global $bad;
    echo "$bad␣and␣$badref\n";
    static $bad;
    echo "$bad␣and␣$badref\n";
}
$bad = 999;
staticgloballocalmess();
echo "$bad\n";
?>
```

Figure 1. Static, local, and global scope ad nauseam

It would take us too far afield to discuss the precise forms of expressions and statements. Suffice to say that both are to a large extent similar to many mainstream languages such as Perl, Java, and C, including the usual loops, conditionals and assignments.

2.0.1 Limitations

PHP is a rather large language and, as our Figure 1 intended to show, it has some obscure features. For both reasons, our analysis and prototype implementation do not cover it completely. These are the limitations we imposed:

- The main restriction is that we do not support classes and objects in our prototype. We found that adding support for these directly involves a lot of work, but conceptually offers little that is new. In our experience as teachers of PHP, only few students define their own classes in PHP. On the other hand, many libraries do depend on them. The `try-catch` construction demands the use of classes, so we omitted those too.
- As the example in Figure 1 shows, global and static declarations can occur anywhere and repeatedly inside a function. Our implementation, however, insists that global declarations inside functions only occur at the start of the function, followed by the static variable declarations, and finally the rest of the statements.
- PHP allows the textual inclusion of code by means of the `include` statement. We demand that arguments to `include` are literal string constants. E.g., `include ("helpers.inc");` is allowed, but `include ($helpers . ".inc");` is not. This obviates performing dataflow analysis to determine possible values for `$helpers`.
- PHP, like Perl, provides support for HereDoc. This facility allows the programmer to escape to text mode, but retaining the ability to interpolate variables. An example is the following:

```
$signature = <<<EOS
best regards,
```

```
$name.
---
Contact me at $mail at gmail.com
EOS;
```

Our tool, however, does not support HereDoc. However, every PHP program with Heredoc can be quite easily converted to a PHP program without (using string concatenation).

- Every case inside a switch must end with a break, so no fall-through is allowed.
- Programmer-defined superglobals are *not* allowed; knowing which exist and when they came into existence is quite complicated, and for the kind of programs we want to analyse, not worth the effort. The superglobals that are part of PHP itself *are* supported.
- Functions may only be declared at top level, e.g., not inside a conditional statement. Function definitions may, however, be inserted by `include` statements.
- The `unset` construct that deletes identifiers is not supported.
- In our type system, callback parameters for functions are treated as strings. We do not interpret the contents of these strings, so when used the functions that are called are not analysed.

From the above list, the restrictions that we believe to have the most practical impact are the omissions of classes and `try-catch`, HereDoc, and the switch statement. Except for the amount of work involved, we believe the concepts can be captured by our approach as well. In the case of classes, we do believe that the performance of the system may strongly degrade. In Section 9 we give some ideas how this degradation in performance may be countered.

```
1  <?php
2
3  function allOne () {
4    $a  = 1;
5    $gbl = $a + 1;
6    $a  = "one";
7    echo $a;
8  }
9
10 function allBoo () {
11   global $gbl;
12   $a  = "boo";
13   $gbl = "boo";
14 }
15
16 $gbl = array("boo");
17 echo count($gbl) . "\n";
18
19 while ($index < count($gbl)) {
20   $entry = $gbl[$index] . "\n";
21 }
22
23 $cnt = $cnt + 1;
24
25 allOne();
26 allBoo();
27
28 echo $gbl;
29 ?>
```

Figure 2. An example piece of PHP code

3. Example

In this section we discuss the results of running our tool on a small piece of PHP code (see Figure 2). For reasons of space, the example is necessarily short. We hope it still gives the reader a flavour of what the system can do.

Consider the results in Figure 3. The system starts by indicating which analyses were applied (this can be changed by modifying a configuration file). It then supplies a list of warnings, each accompanied by a priority. The priorities for the warnings are determined based on settings made in the configuration file. This makes it possible for the user to highlight certain mistakes, by assigning high priorities to them (which for now must be done by modifying the source file `src/php/analysis/typing/TypeInference.java` in the distribution (see Section 6)).

The first warning in Figure 3 arises at line 19, where the system discovers that a variable `$index` is used without having first been assigned to. The second warning addresses the same problem arising at line 20. We have chosen not to collapse similar warnings, for two reasons. First, this is in general not a trivial exercise, and, second, the severity of a potential problem is stressed by having multiple warnings devoted to it.

The system also finds that the (global) variable `$gbl` is assigned values of distinct types, listing the two types in the message. The final warning in the listing can be viewed as an immediate consequence of this problem, pointing to possible places where the multiplicity of types may have consequences. Because the system has evidence that `$gbl` may at some point have an array type, and `echo` needs an argument of type string, there is a potentially lossy conversion from array to string. Our analysis is designed to collapse the types that a global variable may have during execution into a single set of types. This is different for local variables for which we have a set for every program point inside the function where it is used. Therefore, the system cannot discover that in actual fact, everything happens to be fine: during the call to `count` the variable is an array, during the execution of `echo` it is of type string.

The message may be considered a false negative: we get a warning, but nothing is wrong. However, from a didactic point of view, it may be wiser to present the warning anyway. People tend to move around their code, which easily invalidates all kinds of assumptions about when a variable is of a particular type. We think it is wiser to inform the programmer of the fact that a particular global identifier can take on values of different type, so that he can make up his mind whether it would be safer to introduce a new variable for one of its uses instead.

The above reasoning may be defensible for global variables, it certainly is not for local ones. Indeed, our system does not complain about the fact that the local variable `$a` has values of different types, as long as it concerns local variables that belong to different functions. Therefore, it does not complain about the variable `$a` in the function `allBoo`, but it does complain about the type change to the variable `$a` in the function `allOne`.

The final message concerns the fact that a particular variable is used both globally and locally in a function. This warning can be very important to Perl programmers, because they are used to having variables inside a function being assigned global scope by default. Again, the programmer may want to rename one of the two to avoid this sort of clash. If he prefers to keep it this way, he can get rid of the message by suppressing it.

The fact that we discover the undefined use of of `$index` is actually a lucky coincidence; the analysis was not tailored to obtain this kind of result. The reason that it is found at all, is because there is no assignment to `$index` in the program, which makes it undefined throughout the program. Because, as explained above, we store only a single set of types for each global variable, and this

```
Reading file data/paper-example.php
Performing analysis: RE:CODE_GLOBAL_NAMES
Performing analysis: SD:TYPE_ASSIGNS, SD:TYPE_CONDITIONS
Performing analysis: FP:TYPE_INFERENCE

Warnings:
data/paper-example.php:19 Priority: 0.8
Class: FP:TYPE_INFERENCE - NEW_VARIABLE
Message: Undefined variable found: $index
Expected: $index=[num]
Found: $index=[]

data/paper-example.php:20 Priority: 0.8
Class: FP:TYPE_INFERENCE - NEW_VARIABLE
Message: Undefined variable found: $index
Expected: $index=[string, int]
Found: $index=[]

data/paper-example.php:10 Priority: 0.7
Class: FP:TYPE_INFERENCE - MULTI_TYPE_VAR
Message: Multiple possible types found for
         variable $gbl
Found: $gbl=[string, array[string]]

data/paper-example.php:6 Priority: 0.5
Class: FP:TYPE_INFERENCE - TYPE_CHANGE
Message: Type has changed for variable $a
Expected: $a=[int]
Found: $a=[string]

data/paper-example.php:3 Priority: 0.4
Class: RE:CODE_GLOBAL_NAMES - LOCAL_NAME_CLASH
Message: Local variable $gbl in function allOne is
         also used in the main program
Found: local=[$gbl]

data/paper-example.php:28 Priority: 0.4
Class: FP:TYPE_INFERENCE - COERCION_TO_ARRAY
Message: array to non-array coercion for variable $gbl
Expected: $gbl=[string]
Found: $gbl=[string, array[string]]

FP:TYPE_INFERENCE: 5 warning(s)
  TYPE_CHANGE: 1
  COERCION_TO_ARRAY: 1
  MULTI_TYPE_VAR: 1
  NEW_VARIABLE: 2
RE:CODE_GLOBAL_NAMES: 1 warning(s)
  LOCAL_NAME_CLASH_WITH_MAIN: 1

Total warnings found: 6
Total Time used: 0.304s.
Files: 1
```

Figure 3. The result of applying PHP-validator to Figure 2

$$
\begin{array}{lcl}
type & ::= & type\_set \mid \mathbb{P}(function\_type) \\
base\_type & ::= & \mathit{int} \mid \mathit{float} \mid \mathit{bool} \mid \mathit{string} \mid \mathit{resource} \\
 & & \mid \mathit{array}[base\_type] \mid \mathit{array}[\mathit{any}] \\
type\_set & ::= & \mathbb{P}(base\_type) \mid \emptyset \mid \mathit{any} \\
function\_type & ::= & (type\_set)^* \rightarrow result\_type \\
result\_type & ::= & type\_set \mid \mathit{void}
\end{array}
$$

Figure 4. The type language

set will be $\{\mathit{int}\}$ for `$cnt` due to the assignment on line 23, we cannot discern that `$cnt` was used when it was still undefined. As it turns out, the use of `$cnt` even turns out to be compatible with the way it was used. It is as if, from the perspective of the system, the `$cnt` was of type *int* during the whole of the main program.

Although this behaviour may seem a shortcoming at first, changing this has some serious consequences. First, it will make the analysis (much) more costly, because we need to be able to have distinct sets of types for each global variable in each program point. Second, with the current analysis it pays to program cleanly. It is therefore advisable to add a different form of analysis to the system, similar to that employed by Java compilers to determine which assignments are well-defined to discover the undefinedness of `$cnt` in some other way, and not by making the analysis in this paper more refined.

The report generated by our tool concludes with a summary, listing for each type of warning, how many times it occured, and the total number of warnings derived from the code. The system also indicates how long it took to obtain the results.

4. The constraint-based type system

The type system we use is straightforward. We employ constraints to specify the expectations for the types of expressions, and how they should propagate through the program. We benefit here from the fact that most functions and operators do not actually propagate types, in the sense that not the types of the arguments, but the return type of operators and functions determine the type of expressions. It should be noted that it is possible to define parametrically polymorphic functions in PHP, for example an identity function that simply returns its argument. Although our type system cannot infer polymorphic types based on the source code, a programmer may specify such a type for a function and have the system use that instead (see Section 6).

4.1 The type language

We introduce the range of types that we need in order to formulate and implement the soft typing system. The complete syntax for types can be found in Figure 4.

We associate a *set* of types with each identifier, variable or function. There are two kinds: value type sets, $type\_set$, and function type sets, $\mathbb{P}(function\_type)$. The value type sets will be used for ordinary variables, those that start with the symbol $.

A variable that is never used will be associated with the empty set, $\emptyset$; a variable of which no information is known about the types it may take will be associated with *any*. The latter should be understood as representing the set of all (value) types. The base types consist of the usual suspects: *int*, *float*, *bool*, and so on, but also arrays of which the elements are base types, and a special type $\mathit{array}[\mathit{any}]$. As one might expect, the latter type represents an infinite set of types, i.e., the smallest solution to the equation

$$\tau = \{\mathit{array}[\tau] \mid \tau \text{ is a } base\_type\} \ .$$

Functions are associated with a set of function types. Each function type consists of a (possibly empty) list of type sets, one

for each argument, and a result type. The result type is either a simple type set, or the special type $void$ which represents the fact that no value is returned, i.e., the function is a procedure.

From the grammar of Figure 4, we can easily construct a complete lattice of value type sets, based on the partial ordering of subsets on types: $\tau \subseteq \tau'$ if the set of values represented by τ is a subset of that represented by τ'. For example, $\emptyset \subseteq \tau$ for all value type sets τ, $\{int\} \subseteq \{int, float\}$, and $\{int, float\} \subseteq any$. The lattice does not exhibit a *finite height*. An example of an infinite chain is

$$\{array[int]\} \subseteq \{array[int], array[array[int]]\} \subseteq$$

$$\{array[int], array[array[int]], array[array[array[int]]]\}...$$

It is actually quite easy to come up with a PHP program in which a variable actually obtains these types during an (infinite) execution:

```
$a = 1;
while (1 < 2) {
  $a = array($a);
  print_r($a);
}
```

4.2 The constraint language

We formulate our type system by specifying for each expression, or statement, the constraints that should be imposed on the types. The language of constraints we use is maybe larger than expected. This has to do with the fact that in some cases we want to propagate type information from one expression to another: if we add two values, then the type of the expression becomes *num*. On the other hand, if the values we add are non-numerical, then a warning should be issued at this point. Even so, a coercion *does not change the types of those arguments*.

In our specification, we label expressions with unique labels, denoted by $\ell, \ell_1, ...$ A labelled expression or statement is then written e^{ℓ}. During fixed point algorithm, we shall be computing sets of types for all program points ℓ; these sets of types are denoted $type(\ell)$.

The following three basic types of constraints are employed by our constraint-based type system:

$\{\tau_1, ..., \tau_n\} \subseteq type(\ell)$
: This constraint forces the types resulting for the expression e labelled with ℓ to be $\{\tau_1, ..., \tau_n\}$. Often the type set is a singleton, in which case we write $\ell := \tau$ instead. This type of constraint is often used in assignment contexts, or for expressions that have an operator such as + at top-level.

$\ell \equiv \tau$
: The constraint $\ell \equiv \tau$ does not introduce new types for ℓ, but asserts 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 (after fixed point iteration). If more than one expected type is possible for ℓ, we write $\ell \subseteq_{\equiv} \{\tau_1 ... \tau_n\}$.

$\ell_1 \Leftarrow \ell_2$
: This constraint implies a data-flow from ℓ_2 to ℓ_1. In other words, we should enforce that $type(\ell_1) \subseteq type(\ell_2)$. Sometimes, we use the latter notation directly.

For reasons of space, we cannot include the complete collection of type rules for the language, and restrict ourselves to a few illustrative cases. For the remaining rules, the reader may consult Section 7.3 of [4].

Figure 5 contains a number of typical constraint collecting rules for the language. Constraints are associated with statements and expressions alike: $\mathcal{C}^{\star}[e^{\ell}]$ is defined to be the set of constraints contributed by the statement or expressions labelled ℓ, excluding the constraints contributed by any subexpressions; these are added implicitly.

We consider each of the constraint rules in turn. The first is straightforward: whenever we encounter an expression that looks like a floating point number, we insist that the type is *float*. For array indexing, we should verify that v is indeed of array type, containing elements of whatever type. The expression used as index must evaluate to a string or a number. The result type of the expression is the type of the element of the array. This is formulated by by means of a array type deconstructor: $fromarray$, which simply peals off the outer array constructor.

The rule for an explicit type cast is an important one: the type of the result is forced to be the type implied by the cast, but more importantly, there is no expected type constraint. This implies that for the type system a type cast cannot go wrong. The rule allows programmers to insert explicit casts and thereby may get rid off type warnings (see Section 7). The rule for binary boolean operators specifies that the operands should be of type *num*, and that result type must be *bool*. For the conditional expression, the types of the then and else part are propagated to the conditional, and we verify that the type of the conditional expression is indeed boolean.

The while statement is actually quite boring. The only demand we need to make here is that the type of the condition is boolean. Finally, we consider the foreach statement that iterates over an array and binds the contents to the . First of all, the type of e_1 should be consistent with an array type. Second, the type of the identifier here introduced should be set to the element type of the array.

We conclude here by shortly discussing how functions are handled. A function call is straightforward: we simply propagate the types of the argument expressions to the corresponding formal parameter. However, we need to take into account that fewer arguments than necessary may be passed, and that the types of some of the default expressions should be used instead. Each return statement inside a function f propagates the types of the returned expression to a special type set $result_f$, which in turn is propagated back the call sites, that provided the precise sequence of types that led to this particular return set of types. This is handled by means of context, and discussed in the next section.

5. The type inference algorithm

To compute the sets of types for the identifiers, we use a variant of the Maximal Fix Point algorithm, see, e.g., Section 2.4 of [13]. In this particular case, the approach is a form of abstract interpretation with a suitable widening operator to ensure termination [8]. Widening is necessary, because the worst case complexity of the algorithm includes the height of the type lattice, which is infinite in our case.

The analysis can also be viewed as a form of dataflow analysis, propagating and transforming environments (mappings from variables to typesets) along the edges of the flow graph of the program. Indeed, this intuition is a bit closer to the worklist implementation that we use.

5.1 The widening

In dataflow analysis, whenever execution paths join up in the program, the information from different paths is combined into a single analysis value, in our particular case by taking the union of the sets of possible types for each variable. For example, in the statement following a conditional, we compute the set of types for each variable `$var` by taking the union of the types for that variable at the end of the then-part and the end of else-part. If the lattice is of infinite height, it may well be that we have shall have to compute that union an infinite number of types, leading to non-termination

$\mathcal{C}^\star[\texttt{float}^\ell] = \{\ell := \textit{float}\}$ (e.g. $\mathcal{C}^\star[\texttt{6.5}^\ell] = \{\ell := \textit{float}\}$)
$\mathcal{C}^\star[(v^{\ell_1}[e^{\ell_2}])^\ell] = \{\ell \equiv fromarray(\tau) \mid \tau \in type(\ell_1)\} \cup \{\ell_1 \equiv \textit{array}[\textit{any}], \{\textit{int}, \textit{string}\} \subseteq_\equiv type(\ell_2)\}\}$
$\mathcal{C}^\star[((\tau)e^{\ell_1})^\ell)] = \{\ell := \tau\}$ where $\tau \in base\_type$
$\mathcal{C}^\star[\texttt{array}(e_1^{\ell_1}, ..., e_n^{\ell_n})^\ell] = \{\textit{array}[\tau] \subseteq type(\ell) \mid \tau \in type(\ell_i), i \in \{1...n\}\}$
$\mathcal{C}^\star[(e_1^{\ell_1} \oplus e_2^{\ell_2})^\ell] = \{\ell_1 \equiv \textit{num}, \ell_2 \equiv \textit{num}, \ell := \textit{bool}\}$ where $\oplus \in \{$ `<`, `>`, `<=`, `>=`$\}$
$\mathcal{C}^\star[(e_1^{\ell_1}$ `?` $e_2^{\ell_2}$ `:` $e_3^{\ell_3})^\ell] = \{\ell_1 \equiv \textit{bool}, \ell_2 \subseteq type(\ell), \ell_3 \subseteq type(\ell)\}$
$\mathcal{C}^\star[(\texttt{while (}\ e^{\ell_1}\ \texttt{)}\ S^{\ell_2}\ )^\ell] = \{\ell_1 \equiv \textit{bool}\}$
$\mathcal{C}^\ell[(\texttt{foreach}\ (e^{\ell_1}\ \texttt{as}\ v^{\ell_2})\ S)^\ell] = \{\ell_1 \equiv \textit{array}[\textit{any}], \{fromarray(\tau_1) \mid \tau_1 \in type(\ell_1)\} \subseteq type(\ell_2)\}$

Figure 5. The constraint rules specification for a part of PHP

of the analysis. To avoid an infinite chain of ever larger sets we do not simply take the union, but instead apply a widening operator that approximates $\cup$. This corresponds intuitively to taking larger steps in the lattice to ensure that we end up in a solution, i.e., a reductive point, in finite time. The price we pay is that the analysis result we compute is not guaranteed to be a least fixed point, the most precise analysis result that satisfies the constraints. The result does, however, satisfy the constraints.

To determine a suitable widening operator, we first need to answer the following question: how are we going to employ the sets of types that we compute? If the set of types for a variable contains exactly two types, then we probably want to give a warning to the programmer that a type change may occur for that particular identifier, listing those two types explicitly in the message. But what if it concerns ten types, or a hundred? Does it make sense to list them all? We don't think so. At some point, the number of types becomes so large that the types themselves are irrelevant, only the fact that the set of types is too large. This is the key to our choice for a widening function.

First we introduce some auxiliary definitions. The *array depth* of a $base\_type$ is defined as follows:

$$d(\tau) = \begin{cases} d(\tau') + 1 & \text{if } \tau = \textit{array}[\tau'] \\ 0 & \text{otherwise} \end{cases}$$

We extend d to sets of types by taking the minimum depth over all the element types:

$$d(s) = \min\{d(\tau) \mid \tau \in s\}\ .$$

For any constant k, we now define the widening operator ∇_k on sets of value types as follows:

$$\nabla_k(s_1, s_2) = \begin{cases} s_1 \cup s_2 & \text{if } |s1 \cup s2| \leq k \\ \{\textit{array}^{d(s_1 \cup s_2)}[\textit{any}]\} & \text{otherwise} \end{cases}$$

Here, $\textit{array}^0[t] = t$, and $\textit{array}^n[t] = \textit{array}[\textit{array}^{n-1}[t]]$, for $n > 0$. For example, if at least one of the types in $s_1 \cup s_2$ is a non-array type, then the result is *any*, which indicates that we have lost all concrete information about the types of a particular identifier. If, on the other hand, all elements of $s_1 \cup s_2$ are of the form $\textit{array}[\tau]$ for non-array types τ, then we obtain the more precise approximation $\textit{array}[\textit{any}]$. Note that in the latter case, during further iteration we may add new types to the set again. However, adding $\textit{array}[\textit{int}]$ to $\textit{array}[\textit{any}]$ has no effect, so to change the set, only types of lesser depth can be added. This ensures that in the end we either converge or end up in *any*.

Function types are treated differently. For functions we want to discover whether their type is non-deterministic in the following sense: given the types of its arguments, is it possible that the return value of the function is of indeterminate type. We do want to allow that we can derive different return types for different sequences of argument types. Since, by the widening described above, the sets of types for all identifiers are of restricted size, and a function has a fixed number of arguments, we only need to consider a finite number of combinations for each function. Note however that in the worst case this number can be $f * n^k$, where k is the maximum number of types in the type sets, f is the number of functions and n is the maximum number of parameters. In practice, this is not a problem.

5.2 The algorithm

In the master thesis of the first author, the algorithms are introduced step by step. Due to reasons of space, that is not possible here. Therefore, we describe only the initial algorithm without support for functions and context (see Section 2.5 of [13]) and for the different ways of dealing with local, global and static variables. We then describe how these can be added, but do not give the extended algorithms.

The algorithm is a worklist algorithm, a variant of the standard Maximal Fixed Point algorithm (see, e.g., Section 2.4 of [13]). Usually, to each program point two analysis values are associated: the entry value that describes the types of identifiers right before the execution of a statement, and the exit value, that similarly describes the types immediately afterwards. In our case, we add a third environment, which describes the expected types for each variable.

The values that we compute are environments (mappings from variables and functions to type sets) which we denote by $TEnv$. Note that variable names and function names are lexically different, the former always starting with a $-symbol.

The program flow is specified only on *statement level*: the effects of expressions (e.g. assignment subexpressions) are accumulated in their evaluation order, which is from left to right, where inner expressions are evaluated first. These details are handled by the function $transfer\_function$ which is the function that actually enforces the constraints generated for the statement currently under consideration.

During initialization, we start without any variables and the worklist is set to the entry point of the program, F_0.

Then the algorithm starts to iteratively compute better approximations to the sets of types in the program. It does so by taking an arbitrary dataflow-edge from the worklist, and retrieving the entry and exit environments associated with it from the variable $analysis :: P \rightarrow TEnv \times TEnv \times TEnv$ that stores the three environments for each program point. It then makes sure that for each variable the exit sets satisfies the constraints for the statement, given the input sets of types for the variables used in the statement. During this process the types of many variables may change. If the exit environment does not change, then we proceed to the next worklist edge. If it did change, then we should propagate the newly found exit environment to all the statements that follow the current one in the dataflow graph. These nodes are then also added to the worklist for later consideration.

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

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

METHOD: Step 1: Initialization

F := *flow graph from* $P_\star$;
$W.push(F_0)$;
`foreach` p `in` $P_\star$ `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 6. The worklist algorithm for fixed point computation

When the iteration of Step 2 terminates, the worklist is empty, and a solution has been found. Step 3 then verifies which of the expected constraints is inconsistent with the analysis result.

In the next step, we add functions to the process. The main consideration here is that we want to introduce context at this point: Instead of having two environments for each program point, we have two environments for each combination of program point and context. When considering an ordinary intraprocedural flow edge in the iteration step, we simply apply Step 2 for each context value independently. When calling a function we have to make sure that a context shift takes place. Typically, context is chosen to be the set of call strings of a finite length k (typically $k = 3$ during our analysis). These call strings form an abstraction of the call stack, by listing the k most recently called functions. When calling the function p the environments associated with a context value $[q]$, will be associated with the context value $[p, q]$ inside the function. Function return is dealt with similarly, unwinding the call string again.

The final addition is to deal differently with global, static and local variables. The above exposition behaves as if all variables are local; we do something different for global and static variables. Let's first look at global variables. As we said earlier, we have chosen to have only a single set of types for each global variable, as if there is only a single program point, and a single context value for these. This saves a lot of space, but also complicates matters somewhat. What happens if the type set for a global variable changes? Well, potentially all the statements and expressions that use this variable have to be reconsidered. Therefore all these program points must be added to the worklist. Something similar can be done for static variables, except that static variables are associated with a particular function and so we can retrict ourselves to adding program points for statements and expressions inside the function that depend on it.

6. The implementation

We have implemented a prototype which implements soft type inference, under the limitations described in Section 2. The implementation was made in Java; the parser was made using the JavaCC system [14]. Our implementation can be checked out with `subversion` (`http://subversion.tigris.org/`):

```
svn checkout \
  https://svn.cs.uu.nl:12443/repos/php-validator
```

It should be noted that the program does not compile with javac 1.5.x, but it will compile with javac 1.6.x and Eclipse Java Compiler. Further details on compiling and using the program can be found in the `README` file that is part of the check out.

Like any modern language, PHP comes with a large number of built-in functions. For these functions no code is supplied that can be accessed by the analyser. Instead, we have made a list of type signatures for no less than 521 functions, constants and autoglobals available to the programmer. These are by no means all, but we believe that we have covered all of the commonly used ones. A full list can be found and, if the need arises, adapted in the file `data/function-types.types` in the checked out distribution.

For example, for the function `trim` two type signatures are given, as follows:

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	a project by the first author

Figure 7. Additional statistics for each of the test cases

```
trim=[[[string]] => [string],
      [[string],[string]] => [string]]
```

The above line should be read as follows: we associate with `trim` a comma-separated list of signatures, the first of which takes a single string as an argument and returns a string, the second also returns a string, but takes two string arguments. As the reader can verify in the PHP documentation, the function `trim` may indeed take two arguments, but the second one is optional.

A nice additional feature of the function list is that the programmer can specify (polymorphic) type signatures for any function he might write. Our type system cannot discover that a function implements a polymorphic function, e.g., the identity function. However, the programmer may add a function with its polymorphic type to the function list, so that it can be used properly during type during analysis.

A second configuration file `data/analysis-example.txt` exists, in which the programmer specifies various parameters for running the system, e.g., which analyses should be executed. Inspired by the Valgrind framework [12], it is possible to specify here that certain warnings for a particular analysis should be suppressed. The suppression may apply to the warning as a whole, or it may apply to particular identifiers only. For example, the configuration file may contain

```
...
analysis=FP:TYPE_INFERENCE
suppress=COERCION_NUM
suppress=COERCION_TO_ARRAY for $ar

analysis=RE:CODE_GLOBAL_NAMES
...
```

Finally, we would like to remark that the tool provides a large number of additional analyses for PHP, not discussed in this paper. Examples are the validation of correct use of `$_GET` and `$_POST` and checking for various coding style violations. Moreover, all the analysis are formulated within a framework that is independent of the language under analysis. More details can be found in the master thesis of the first author [4].

7. Experimental results

We have applied our tool to four collections of PHP programs, written by people of varying proficiency. We first describe the various sets of test programs, and then discuss the results (i.e. the total number of warnings generated) and the validity of these numbers.

We have tested our analyses on the following sets of programs (see Figure 7 for some statistics for each of the cases):

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 assignments of approximately 30 groups have been analysed.

Analysis / warning	Case 1	Case 2	Case 3	Case 4
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

Figure 8. For each test case, the numbers of warnings for each analysis specified by kind

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.

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, although 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 has been in use for some time and was written by the author of this thesis.

In Figure 8 the test results for these analyses on our test cases are displayed.

It appears that the final practical assignments (Cases 2 and 3) generate fewer warnings on average than the first one (Case 1). This may be due to the relative inexperience with programming PHP at that point.

There are some interesting cases that seem rather inaccurate or differ a lot from the other numbers. Consider the `NEW_VARIABLE` warning type, which indicates undefined variables or variables of which the initial type cannot be determined. Inspection of the source code shows that these occur mainly when a web input (or session) variable is assigned to a PHP variable. Since the external values are of type any, this leads to propagation of unknown type information.. Explicit coercions easily 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 assigned type *any*, 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]
```

An explicit coercion `(int)$row[product_id]` in the call to `querySpecialPrice` would solve the problem. Other type problems such as these can be solved similarly. It should be noted that adding explicit casts forces the programmer to think of the type of the value he expects to obtain. Moreover, the explicit type cast also serves to document his expectation explicitly.

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, but which are 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.

An 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 `$num_posts` variable is assigned type `string`, because the `$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` solves this problem.

In the output we tend to obtain quite a few falsely negative `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 (discussed in Section 6), which leads to this warning. 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, then 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 combinations of 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 makes determining the correct result type for specific argument types hard or impossible.

Type changes of a variable are detected well, 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 $string$. Therefore, `$internal` is also assigned type *string*. However, in the conditional statement, the type of `$internal` is changed to *int* in 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 generated by this single example is the `COERCION_BOOL` warning, because the string `$internal` is used in the boolean condition.

The `MULTI_TYPE_VAR` warning occurs when the type of a variable is 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 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]
```

Finding out the exact cause of messages such as these that are caused by this example can be very tricky, especially if function parameters have multiple types. Therefore, programmers should take extra care in determining the result types of their functions. The `FUNCTION_MULTI_TYPE` warning that this example also throws, is a good indication that something is not right. In the above example, the function can return both an integer or a string as result, making its result type $\{string, int\}$.

In general, the type inference algorithm performs quite well. Still, there is enough space for improvement, especially on the de-

tection 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 is in 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).

8. Related work

Although we are not aware of work in soft typing specifically for PHP, many other dynamically typed languages have been considered for soft typing.

For Scheme a soft typing system called Soft Scheme [15] has been developed. This system is modeled after the work of Cartwright and Fagan [6], which is basically the Hindley-Milner system extended with union types and recursive types. It uses an efficient representation for types, integrates 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 [1] programs, and prints a list of the inserted run-time checks, after which the programmer can then inspect type information.

For Perl a static type inference system [11] 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. No actual Perl source code is considered. The type system is a static type inference system which uses a unification algorithm similar to the one used by Damas and Milner [9]. A soft typing system with union types like described in [2, 6] might have been considered here, but instead, they opted for a solution [11] where type variables can be entered in certain places, so that things like unqualified numbers, generic scalars, and references to anything can be expressed without the need for expensive union types. This however resulted in a type language which seems a lot more complicated than usual. On the other hand, the added flexibility is expressive enough to yield the same results as union types would give, and is also more efficient.

For Python, several soft typing systems have been developed. We discuss two important ones. Iterative type analysis [7] 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 specific point of execution in the program. Another type system described in [7], 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 [5].

Aggressive type inference [3] makes use of the idea that not all programs in a dynamically-typed language need dynamic typing. 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 not as accurate as our approach when it comes to local variables.

9. Conclusions and future work

We have described a system that applies soft typing to PHP. The goal of the system is to deliver a list of warnings, each accompanied by a priority that indicates the seriousness of the problem. The system is not supposed to be sound or complete, both false negative and false positives may occur. To deal with false negatives, we have added a facility for suppressing warnings.

An obvious extension of our work is to incorporate the missing parts of PHP (and maybe even later versions of the language). It is quite possible that the addition of classes, incurs a large penalty on the run-time of analyses, so it may be necessary to improve the efficiency of the implementation. This may either be by careful programming, by restricting context further, or the optimization discussed below. A second useful addition is to implement a full definite assignment analysis to track down the missing *undefined variables* such as `$cnt` in Section 3. Another extension would be to perform an additional check on type casts, to find out whether the same variable is explicitly cast to two different types.

In the opposite direction, we would be interested to compare our current version with a variant in which global variables are analysed with some measure of context. An additional benefit of this more uniform treatment of global and local variables is that the workflow algorithm does not need to explicitly distinguish between them. The algorithm will therefore become much simpler. We do believe however, that the run-time penalty will be quite large.

A characteristic of our worklist algorithm is that it if the analysis result for a *particular context value* changes, then the analysis results for all context values will be repropagated. This can be avoided by annotating each value for a particular context value with a change bit, so that we propagate only those values that have actually changed. We did not implement this, because increased efficiency is currently not important. However, such an optimization might allow using more context information and potentially higher precision, and allow us to deal with classes effectively.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams, IV, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, 1991.

[2] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[3] J. Aycock. Aggressive type inference. In *The Eighth International Python Conference*, 2000. `http://www.python.org/workshops/2000-01/pr`

[4] P. Camphuijsen. Soft typing and analyses on PHP programs. `http://www.cs.uu.nl/wiki/Hage/MasterStudents`.

[5] B. Cannon. *Localized type inference of atomic types in Python*. PhD thesis, California Polytechnic State University, 2005.

[6] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI '91)*, pages 278–292, 1991.

[7] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. *Lisp Symb. Comput.*, 4(3):283–310, 1991.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[9] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

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

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

[12] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007. `http://valgrind.org/`.

[13] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, second printing edition, 2005.

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

[15] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, 1997.