An executable formal semantics of PHP

Daniele Filaretti and Sergio Maffeis

Department of Computing, Imperial College London
{d.filaretti11,sergio.maffeis}@imperial.ac.uk

Abstract. PHP is among the most used languages for server-side scripting. Although substantial effort has been spent on the problem of automatically analysing PHP code, vulnerabilities remain pervasive in web applications, and analysis tools do not provide any formal guarantees of soundness or coverage. This is partly due to the lack of a precise specification of the language, which is highly dynamic and often exhibits subtle behaviour.

We present the first formal semantics for a substantial core of PHP, based on the official documentation and experiments with the Zend reference implementation. Our semantics is executable, and is validated by testing it against the Zend test suite. We define the semantics of PHP in a term-rewriting framework which supports LTL model checking and symbolic execution. As a demonstration, we extend LTL with predicates for the verification of PHP programs, and analyse two common PHP functions.

1 Introduction

PHP is one of the most popular languages for server-side scripting, used by amateur web developers as well as billion-dollar companies such as Google, Facebook and Yahoo!. It is used for developing complex programs, enabling all sort of sensitive activities such as online banking, social networking, and cloud computing. Despite the flexibility and ease of use of PHP, its dynamic features (shared by similar scripting languages) make it easy to introduce errors in programs, potentially opening security holes leading to the leakage of sensitive data and other forms of compromise.

Many web applications have reached a level of complexity for which testing, code reviews and human inspection are no longer sufficient quality-assurance guarantees. Tools that employ static analysis techniques [9,34,20] are needed in order explore all possible execution paths through an application, and guarantee the absence of undesirable behaviours. However, due to classic computability results, this goal can be accomplished only by applying a certain degree of abstraction, with consequent loss of precision (i.e. introducing false positives). To make sure that an analysis captures the properties of interest, and to navigate the trade-offs between efficiency and precision, it is necessary to base the design and, we add, the development, of static analysis tools on a firm understanding of the language to be analysed.

The main contribution of this paper is to present KPHP, the first formal (and executable) semantics of PHP, which can serve as a basis to define program analyses and semantics-based verification tools (Section 3).

Some programming languages, such as Standard ML [30], already come with a formal specification. Others, such as C [19] and JavaScript [18] are specified in English prose with varying degrees of rigour and precision, and have recently been formalised [32,25,3,15,11,4]. PHP is only implicitly defined by its *de facto* reference implementation (the Zend Engine [14]), and the (informal) PHP reference manual [13]. Due to the lack of a document providing a precise specification of PHP, defining its formal semantics is particularly challenging. We have to rely on the approximate information available online, and a substantial amount of testing against the reference language implementation. We do not to base our semantics on the source code of the Zend Engine in order to avoid bias towards inessential implementation choices. In defining our semantics, we identify several cases where the behaviour of PHP is complicated and unexpected. Some of these examples are known to PHP programmers, and have contributed to driving our design. Other examples are new, and were discovered by us as a consequence of semantic modelling (Section 2). Although useful to get introduced to each language construct, the online PHP language reference [13] is not precise enough to serve as a basis for a formal semantics. Quite the opposite, we hope that our formal semantics may serve as a basis to create a precise, English prose specification of PHP in the style of the ECMA specification of JavaScript [18].

We write our semantics in K [39,37], a framework for defining programming languages on top of the Maude [8] term-rewriting tool. A language semantics as expressed in K has a rigorous meaning as a term rewriting system, and is suitable for formal reasoning and automated proofs. Moreover, it is directly executable, enabling a tight design-test loop which is crucial for the test-driven semantics development needed in the case of PHP. Extensive testing using official test suites is becoming "best practice" to validate executable semantics of programming languages [15,11,4]. We validate KPHP by automated testing against the Zend PHP test suite [40] (Section 4), and we design additional PHP tests in order to cover all of the semantic rules, including those not exercised by [40].

The main goal of our semantics is to provide a formal model of PHP upon which semantics-based verification tools (such as abstract interpreters, type systems and taint-checkers) can be built. Developing such tools goes beyond the scope of this paper. However, we are able to begin demonstrating the practical relevance of KPHP by using it for program verification. In particular, the K framework exposes Maude's explicit-state Linear Temporal Logic (LTL) model checking to the semantics [11], and supports symbolic execution for any language definition [1]. We define an extension of LTL with predicates to express interesting temporal properties of PHP programs, and verify two representative PHP functions from `phpMyAdmin` [33] and the PHP documentation [10] (Section 5).

An extended version of this paper, together with the latest version of the semantics, and all the KPHP development (including KPHP interpreter, tests and verification examples) is available on `http://phpsemantics.org`.

2 A PHP Primer

In this Section, we give a brief introduction to the PHP language and its usage, and present examples of some challenging and surprising features of the language. [1] Some of these examples are known to PHP programmers, and have contributed to driving the design of our semantics. Others are new, and were discovered by us, as a consequence of semantic modelling.

Hello World Wide Web. PHP scripts are typically run by web servers. Typing a URL such as `http://example.com/hello.php?name=xyz`in a browser may cause the responding server to invoke PHP on the file `hello.php` listed below:

```
<? echo "<HTML><Body>Hello␣".$_GET["name"]."!</Body></HTML>"; ?>
```

This minimal example illustrates the typical behaviour of a PHP script. It receives inputs from the web and it responds by generating an HTML page depending on such inputs. The predefined `$_GET` array is in fact populated from the parameters of the HTTP request, and `echo` is a simple output command that in shell mode prints to standard output but that in server mode generates the body of the HTTP response message. In this paper, we focus on PHP as a programming language, and leave the important topic of formalisation of the server execution model to future work.

2.1 PHP: a Closer Look

We now describe some features of the core PHP language which may be unfamiliar to programmers used to different languages, challenging to represent in an operational semantics, or both.

Aliasing and references. PHP supports variable aliasing via the *assignment by reference* construct. This mechanism provides a means of accessing the same variable content by different names.

```
$x = 0;
$y = &$x;        // $x and $y are now aliased
$y = "Hello!";
echo $x;         // prints "Hello!"
```

Aliasing can be useful for example to write functions that operate on parameters containing large data structures, avoiding the overhead of copying the data structure inside the local scope of the function. On the other hand, aliasing is notoriously difficult to analyse statically. PHP references are different from pointers (as in C) in that neither address-arithmetic nor access to arbitrary memory is allowed. For example, the following code would be rejected:

```
$x = (&$x + 1); // causes a parse error
```

[1] All the examples are reproducible by pasting the code in the PHP Zend Interpreter (version 5.3.26 or similar) available in most OSX or Linux distributions. The symbol > precedes the shell output. For readability, here we re-format the output of `var_dump`.

Braced and variable variables. The official PHP documentation gives the following description for *variable variables*: "A variable variable takes the value of a variable and treats that as the name of a variable". Here is an example:

```
$x = "y";
$y = "Hello!";
echo $$x;                // prints "Hello!"
```

Hence, in a PHP semantics, variable names should be modelled as a set of string-indexed constructors, rather than as a set of unforgeable identifiers.

Variable variables are useful for example to simulate higher-order behaviour by passing functions *by name*. On the other hand, they hinder static analyses, because it is not possible in general to determine statically the set of variables used by a PHP script. A similar argument applies to braced variables, a syntax to turn the result of an arbitrary expression into an identifier, as for example in

```
${"x"} = "y";            // defines variable $x
$z -> {"x".$x};          // access field xy of object $z
```

Type juggling. Each PHP value has a type (`boolean`,`integer`,...). Automatic type conversions are performed when operators are passed operands of the incorrect type. For example, non-empty strings are translated to the boolean `true`, and booleans `true` and `false` are converted respectively to the integers `1` and `0`.

```
if ("false") echo true + false; else echo "false"; // prints "1"
```

Some type conversions need to be defined explicitly. For example, an object can be converted to a string by defining the *magic method* `__toString`. If such method is undefined, the attempted conversion triggers an exception.

Type juggling makes it easier to write code that does not get stuck, but also increases the probability that such code will not behave as expected. For example, although the conversion of objects to numbers is undefined according to the online documentation, the Zend engine converts objects to the integer `1` (our semantics mimics this behaviour, and issues an additional warning).

Arrays. Arrays in PHP are essentially *ordered* maps from `integer` or `string` *keys* to language values. If a value of a different type is given as a key, type juggling will try to convert it to an `integer`.

```
$x = array("foo" => "bar",4.5 => "baz");
```

The array `$x` above maps `"foo"` to `"bar"` and `4` to `"baz"`. Note how the `float` value `4.5` was automatically converted to the `integer` value `4`.[2] Array elements can be accessed via standard square-bracket notation, and it is also possible to assign an element to an array without specifying a key.

```
$x[] = "default"         // use default key 5
$echo x[5]               // prints "default"
```

[2] Although we model array key conversions in our semantics, we do not give full details about them here. The interested reader can try evaluating this: `$x = array( 1=>"foo", "2"=>"wow", 3.5=>"doh", "4.5"=>"omg", NULL=>"lol");`.

In this case, a default key (the greatest integer key already defined, plus one) is used. Arrays contain an internal pointer to the *current* element (the first by default), which can be manipulated using functions `current` `next`, `each` and `reset`:

```
echo current($x);        // prints "bar"
next($x);                // advances the pointer
echo current($x);        // prints "baz"
```

Objects. From a semantic standpoint, PHP objects can be seen as string-indexed arrays with additional visibility attributes (`public`, `protected` or `static`), and with methods inherited by their defining class. Just like arrays, (`stdClass`) objects can be initialised "on the fly":

```
$obj -> x = 0;
var_dump($obj);
> object(stdClass)#1 (1) { ["x"]=> int(0) }
```

Access to an array element is always granted, whereas access to an object property is regulated by the visibility attribute, and depends on the context whence the property is being accessed. Inheritance is class-based. Consider the following example from the Zend test suite [40]:

```
class par {                    class chld extends par {
  private $id = "foo";           public $id = "bar";
  function displayMe() {         public function displayHim() {
    echo $this -> id; }}           parent::displayMe(); }}

$obj = new chld();
$obj -> displayHim();     // prints "foo"
```

Crucially, this code returns `"foo"` because `$id` is declared `private` in the superclass `par`. If instead `par` defined `$id` as `public`, the code would return `"bar"`. In Section 3.2, we shall see how we capture this subtlety in our semantics by indexing the arrays of object fields by *key-visibility* pairs. A notable difference between objects and arrays is that objects are copied by reference whereas arrays are copied by value. Most existing analysis tools for PHP do not support objects, because their semantics is not easy to analyze.

2.2 PHP: Digging Deeper

We now look more in depth, to uncover difficult "corners" of PHP. While the first example below on array copy is a well-known PHP issue [41], the others are our original observations, discovered while developing the relevant semantics rules. Although some PHP experts may be aware of these cases, they are not part of the mainstream knowledge about PHP, and are hence worth discussing.

Array copy semantics. In PHP arrays are copied by value. For example, the code below copies each element of the array stored in `$x` into a fresh array to be stored in `$y`, and then updates the first element of `$x`, without affecting `$y`:

```
$x = array(1, 2, 3);
$y = $x;
$x[0] = "updated";
echo $y[0];              // prints 1
```

Yet, in PHP it is possible to alias a variable to a particular array element. If such sharing happens *before* the array copy, its semantics become quit subtle. Consider the following code:

```
$x = array(1, 2, 3);
$temp = &$x[1];          // we introduce sharing
$y = $x;                 // and assign normally
$x[0] = "regular";       // update a regular element
$x[1] = "shared";        // update the shared element
```

```
var_dump($x);
> array(3) {
    [0]=> string(7) "regular"
    [1]=> &string(6) "shared"
    [2]=> int(3) }
```

```
var_dump($y);
> array(3) {
    [0]=> int(1)
    [1]=> &string(6) "shared"
    [2]=> int(3) }
```

These results show that array `$x` is copied element by element in `$y`, so that the assignment to `$x[0]` affects only `$x`, *except* for the aliased element `$x[1]`, which is now shared with `$y`, which therefore also sees the side effects of the second assignment. Accordingly, our semantics copies the shared elements of the array by reference, and the non-shared elements by value. If a non-shared element is an array itself, the process continues recursively. Matters get even more complicated when taking into account the *copy-on-write* semantics of PHP arrays, as shown by Tozawa *et al.* [41], who first identified this problem and pointed out inconsistencies in the Zend implementation.

Global variables as array properties. In PHP, *global* variables are visible at the top level, and can be imported in functions explicitly using the `global $x;` command. *Superglobals* are special variables directly accessible inside any scope that does not shadow them. Shadowing occurs for example when a function defines a parameter or a local variable with the same name as the superglobal. The superglobal variable `$GLOBALS` points to an array whose properties are the global variables, so that effectively these can be manipulated with the dual syntax of variables or object properties. For example,

```
$GLOBALS["x"] = 42;
echo $x;                 // prints 42
```

Because of this ambivalence of global variables, in the semantics it is natural to model scopes as heap-allocated arrays, rather than as frames of a stack independent from the heap. This is analogous to what happens in JavaScript semantics [25,4], where global variables are the properties of the global object, and scopes are heap-allocated objects. Maffeis *et al.* [27,26] show that confusing variables (which can usually be identified statically) with object properties (which can be computed at run-time) complicates security analyses for JavaScript. The

case for PHP is even more desperate, as "thanks" to variable variables even variables on their own cannot be determined statically.

Evaluation order. In C, the evaluation order of expressions is undefined. In most languages, it follows a left-to-right order. Let us see what happens in PHP.

```
$a = array("one");
$c = $a[0].($a[0] = "two");
echo $c; // prints "onetwo"
```

```
$a = array("one");
$c = ($a[0] = "two").$a[0];
echo $c; // prints "twotwo"
```

This example suggests that the operands of the string concatenation operator "." are indeed evaluated left-to-right. That it should be so easy! Let us see what happens if the operands are simple variables instead of array elements:

```
$a = "one";
$c = $a.($a = "two");
echo $c; // prints "twotwo"
```

```
$a = "one";
$c = ($a = "two").$a;
echo $c; // prints "twotwo"
```

Both print `"twotwo"`, contradicting our hypothesis: the example on the left suggests that expressions are evaluated right-to-left. Through the lenses of our formal semantics, we can explain this behaviour: the arguments to binary operators are indeed evaluated left-to-right, but while evaluating array elements (or object properties) yields the corresponding value, evaluating simple variables yields a pointer that is dereferenced only when the value is effectively needed. In this case, the value of `$a` to the left of the string concatenation is read only after the assignment to the right has taken place.

Object and array iteration. Scripting languages such as JavaScript and PHP provide constructs to iterate over all the properties of an object. Such constructs can be tricky to implement, and hard to model in a formal semantics. For example, the JavaScript formalisation of [4] does not model the for-in loop, as the corresponding ECMA5 standard is inconsistent when it comes to describe object updates within the loop. We discovered that the corresponding `foreach` statement in PHP is also very challenging, due to the presence of aliasing and the behaviour of the explicit `current` pointer.

Consider this example, where we iterate twice through the fields of array `$a`:

```
$a = array('a', 'b', 'c');
foreach ($a as &$v) {};          // aliasing on $v
foreach ($a as $v) {};
```

Since there is no code inside the bodies of the two loops, we could expect the array to remain unchanged. However, a call to `var_dump($a)` shows that that is not the case:

```
array(3) { [0]=> string(1)  "a"
           [1]=> string(1)  "b"
           [2]=> string(1)  "b" }
```

This is what happens: variable `$v`, introduced by the first `foreach`, has global visibility; at the end of the first `foreach`, `$v` and `$a[2]` are aliased; at every

iteration of the second `foreach`, a simple assignment `$v = $a[...]` is made, storing the current array element in `$v`, and hence in `$a[2]`.

There is even worse. Consider the code below: it initialises two objects, creates an aliasing to `$obj1` and iterates on its fields. When the current element is 1 (at the first loop iteration), it replaces the object being iterated upon.

```
$obj1 -> a = 1; $obj1 -> b = 2;
$obj2 -> a = 3; $obj2 -> b = 4;
$ref = &$obj1;           // aliasing on $obj1
foreach ($obj1 as $v){ echo "$v,"; if ($v === 1) $obj1=$obj2; };
if ($obj1 === $obj2) echo "true";
```

This code outputs `1,3,4,true`: the iterator is swapped, and iteration continues on the second object. But if we remove the aliasing on `$obj1` by commenting out the 3rd line, then the output surprisingly becomes `1,2,true`, where the update to `$obj1` is visible only at the end of the loop.

In absence of aliasing, the `foreach` on arrays is analogous. In presence of aliasing instead `foreach` behaves differently. Consider the code below:

```
$a1 = array(1,2); $a2 = array(3,4);
$ref = &$a1;             // aliasing on $a1
foreach ($a1 as $v){ echo "$v,"; if ($v === $a1[0]) $a1=$a2; };
```

This code enters an infinite loop outputting `1,3,3,3,3...`. Since PHP arrays are copied by value (as opposed to objects which are copied by reference), the assignment `$a1 = $a2` copies the whole data structure associated to `$a2`, including its `current` pointer, which is used to perform the iteration, causing the infinite loop (see Section 3.3 for more details on array assignment).

To be precise, during array copy, the original `current` pointer is copied to the new array only if it points to a valid element. If instead the original `current` is overflown, the `current` pointer of the new copy is reset:

```
$x = array(0,1);         // initialise $x
next($x); $y = $x;       // increment current & copy array
echo current($y);        // prints 1 (current was copied)
next($x); $z = $x;       // overflow current & copy array
echo current($z);        // (1) prints 0 (current was reset)
echo current($x);        // (2) prints "" (current is overflown)
```

Once again, we find some erratic behaviour of PHP: if we comment out the output line marked with (1) above, the `current` of `$x` is reset instead, and line (2) prints 0. We consider this to be a bug in the current version of PHP,[3] and our semantics prints `""` for line (2) in both cases (see Section 3.3).

3 $\mathbb{K}$PHP

In this Section, we describe our formalisation of the operational semantics of PHP. Our semantics is vast (more that 800 rules), so we can only provide a

[3] This behaviour is related to PHP Bug 16227, which was resolved in PHP 4.4.1 and was reintroduced since PHP 5.2.4.

roadmap through it, selectively explaining some of the crucial features, and referring the reader to `http://phpsemantics.org` for the gory details.

3.1 Preliminaries: the $\mathbb{K}$ Framework

We write our semantics in $\mathbb{K}$ [39,37], a framework for defining programming language semantics on top of the Maude [8] term-rewriting tool. We chose $\mathbb{K}$ for three main reasons: (i) a semantics in $\mathbb{K}$ has a rigorous meaning as a term rewriting system, supporting fully formal proofs; (ii) a semantics in $\mathbb{K}$ is directly executable, enabling a tight design/test loop; (iii) once a semantics is defined, the $\mathbb{K}$-Maude toolchain provides automatic support for model checking and symbolic execution of programs.

In order to model a language in $\mathbb{K}$, the first thing to be defined is a *configuration*. Intuitively, configurations specify the structure of the *abstract machine* on which programs written in the language will be run, and are represented as labeled, possibly nested multisets, called *cells*. Cells contain pieces of the program state, such as the program to be evaluated, the heap, and function and class definitions. A $\mathbb{K}$ semantic rule for assigning a value to a variable in a simple imperative language looks like this:

$$\left\langle \frac{\texttt{X = V}}{\cdot} \ \ldots \right\rangle_{\texttt{k}} \langle \ldots \ \texttt{X} \mapsto \texttt{N} \ \ldots \rangle_{\texttt{env}} \left\langle \ldots \ \texttt{N} \mapsto \frac{\_}{\texttt{V}} \ \ldots \right\rangle_{\texttt{store}}$$

In this simple case, the configuration has three cells, `k`, `env` and `store`. The `k` cell, by convention, always represents a list of computations waiting to be performed, where the left-most element is a local rewriting rule stating that, after execution, command `X=V` should be removed from the stack ("." is the empty list or set). The rest of the program is denoted by the *don't care* notation "..." for lists. The `env` cell maps variables to locations, and it is used to find the address of `X` via pattern matching. Finally, the `store` cell maps locations to values. The rule above says that if location `N` can be found, its content is overwritten by `V` ("_" matches any term). $\mathbb{K}$ rules can be quite compact and modular, as they only need to mention the cells relevant to the rule at hand. If the only cell needed by a rule is `k`, it can be omitted. For example we can write directly $\frac{\texttt{X = V}}{\cdot}$ instead of $\langle \frac{\texttt{X = V}}{\cdot} \ \ldots \rangle_{\texttt{k}}$.

To control the evaluation order of sub-expressions, rules can be annotated with *strictness* and *context* information. For example, one can write

```
syntax Stmt ::= Id "=" Exp [strict(2)]
```

meaning that `Exp` (meta-variable 2 of the production) is meant to be evaluated before the assignment takes place. Hence, `Exp` is placed at the front of the execution list in the `k` cell, as in $\langle \texttt{Exp} \curvearrowright \texttt{Id} = \square \ \cdots \rangle_{\texttt{k}}$, where $\curvearrowright$ denotes the sequential composition of tasks to be performed, and $\square$ is a place holder that will be replaced by the result of evaluating `Exp`. Further details of the $\mathbb{K}$ framework will become apparent as we describe our semantics.

3.2 $\mathbb{K}$PHP Overview

Parsing. For parsing, we use the PHP grammar from `PHP-front`, a package for generating, analysing and transforming PHP code used by the `PHP-sat` [7] project. The grammar is in the *SDF* [17] format, which can be passed to the `sdf-2-kast` tool [6], which generates an abstract syntax tree for $\mathbb{K}$.

Values. PHP supports three categories of data types: *scalar*, *compound* and *special*. Scalar types are the `boolean`, `integer`, `float` and `string` types, compound types are the `array` and `object` types, and special types are the `NULL` and `resource` types. The `resource` type is used for files and other external resources, and is left for future work. For simplicity, we model scalar types by the corresponding built-in types of $\mathbb{K}$, although there may be some subtle differences for example in the approximations made by floating-point computations.

Arrays and objects. We model arrays as pairs `array(C,EL)` where `EL` is a list of array elements and `C` is an optional *current element*. Array elements are represented as triples `[k, v, l]` where `k` is an `integer` or `string` *key*, `l` is the memory location where the actual value is stored, and `v` a *visibility* attribute (discussed below). Objects are triples `OID(L,CL,ID)` where `L` is the location of an array containing the fields of the object, `CL` is the name of the object's class and `ID` is a unique numeric object identifier. Classes are 4-tuples `Class(SC,IV,MT,SV)` where `SC` is the name of the superclass, `IV` is the list of instance and static variables, `MT` is the method table, and `SV` is a pointer to the scope holding static variables (those shared across all objects from the given class).

The visibility attribute is always `public` for *proper* array elements, whereas it can be also `protected` or `private` for objects fields. As implied by the object inheritance example of Section 2.1, the correct handling of visibility of object properties is subtle. In particular, array elements are identified by a combination of the key and the visibility attribute. In that example, the field array of `$obj` will contain two separate entries `["id",public,l1]` and `["id",private(par),l2]`, necessary to resolve the right element depending on the context. The modelling of arrays, especially when considering the interaction with other features such as aliasing, was one of the main challenges of this and related efforts (e.g. [21,43]).

Finally, the `NULL` value is the value returned when attempting to read any non previously initialised variable, array or object elements.

Values in memory. Following the online documentation, in the memory we wrap values into four-tuples `zval(Value, Type, RefCount, Is_ref)`. Each `zval` contains the value, its type, a reference counter keeping track of the number of locations currently pointing to the value, and a boolean flag indicating whether or not the value is aliased.[4] We define a number of internal low-level operations which manipulates zvals (`zvalRead`, `incRefCount`, etc.), and use them as building blocks for defining higher level functions (`read`, `write`, etc.) providing the illusion of operating directly on simple values, increasing the modularity of the semantics.

[4] The flag `is_ref` is used for implementing the array *copy-on-write* optimisation. We include it just for completeness, since in our semantics, `is_ref` is true iff `refcount`$>$1.

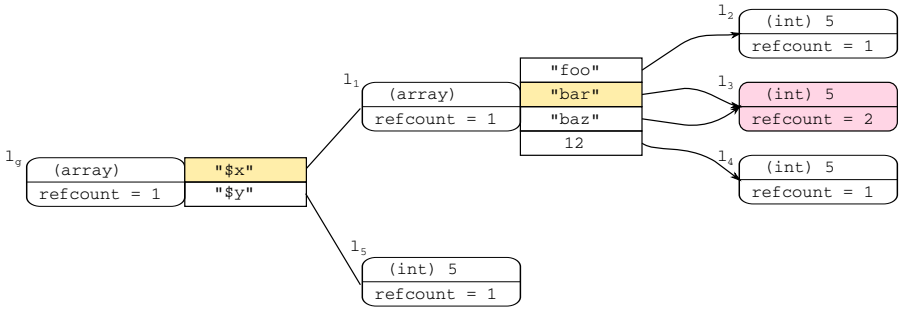

Fig. 1: Example heap, where the reserved location l_g contains the global scope.

Memory. The heap, which is contained in the `heap` cell, is a map $\mathbb{H} : loc \to zval$, where loc is a countable set of locations $l_1, \ldots, l_n$. Fig. 1 shows the heap after executing the program

```
$x = array("foo" => 5, "bar" => 5);
$y = 5;
next($x);
$x["baz"] = &$x["bar"];
$x[12] = 5;
```

where the elements pointed to by the array *current* pointers are shaded (in yellow), and shared zvals are shaded (in red), and l_g is the location containing the global scope, which is a special array where both `$x` and `$y` are defined. We have shown in Section 2.2 how the global scope can be accessed directly via the variable `$GLOBALS`. In fact, just like in JavaScript, it is convenient to represent all PHP scopes as heap-allocated arrays.

References. Several programming languages internally use references, and so does PHP, but with its own original twist. Consider running a simple program `unset($y)` on the state shown in Fig. 1. If the argument `$y` were to be evaluated to a value, we would reach `unset(5)` which is nonsensical. Even evaluating `$y` to the location l_5 would not be the right choice, since in that case we could successfully free the location, but not remove the link from `$y` to l_5 (which is stored in the array at l_g). This is just an example of a general class of cases, which imply that variables need to be evaluated to references of the form `ref(L, K)` where `L` is the address of the array or object containing the variable, and `K` is the variable name. When the actual value stored in the variable is needed, further steps of reduction can be taken to resolve the reference. This is not a trivial process, as the lookup depends on whether the reference appears on the left or right hand side of an assignment. Consider the code

```
$x = $y;
```

where neither `$x` nor `$y` have been initialised. The first step is to evaluate the variables obtaining the references `ref(`l_g`,"x")` and `ref(`l_g`,"y")`. On the left-hand

side, since "x" is not an entry of the (array) scope l_g, it will be created, adding a link to a fresh location. For the right-hand side, since "y" is also not present in l_g, NULL will be returned and written to the fresh location.

Unfortunately, since arrays are copied by value, this is not the end of the story. Consider the following program (adapted from a Zend test):

```
function mod_x() {
        global $x;
        $x = array('a','b');
        return 0;
}
```

```
$x = array(1, 2);
$x[0] = mod_x();
var_dump($x);
>array(2) { [0]=> int(0)
            [1]=> string(1) "b" }
```

If $x[0] was evaluated to a reference before calling mod_x (as in JavaScript), it would become ref(L1,0), where L1 is obtained by resolving the reference ref(L,"x") in the current scope L. Hence, the assignment would affect the original array and the output would still show "a" (instead of 0) at position 0. In order to model the observed PHP behaviour, we introduce a more general type of reference (lref) which can be thought as a "path". In the example above, the expression $x[0] effectively evaluates to lref(ref(L,"x"),0), a value which represent a path starting at the current scope L and ending at the desired location.

The lref mechanism is also fundamental to handle assignments to arrays and objects created on-the-fly. Assume that variable $y is undefined. Consider:

```
$y[] -> x = 42;
```

This is indeed valid PHP code, that creates an array and an object on the fly, adds the object as element 0 to the array, and adds 42 as field x to the object.

Exceptions. The treatment of exceptions is based on an exceptionStack cell where we push the catch branch and the program continuation. If an exception is thrown, the catch is executed, otherwise the continuation is executed.

HTML. In general, a PHP script can be an HTML document that contains several PHP tags <? ... ?> (or <?PHP ... ?>) delimiting regions of PHP code that are executed as part of the same script. The HTML is treated as part of the program output, in the order it is encountered. Our semantics implements this behaviour. Hence, the hello.php example of Section 2 is equivalent to

```
<HTML><Body><? echo "Hello␣".$_GET["name"]."!"; ?></Body></HTML>
```

Configuration. The global configuration of PHP, which represents the global state of the abstract machine, consists of 42 cells, and is shown in Figure 2 (we hide some of the nested cells to improve readability). The script cell contains details of the script being executed, and in particular a cell k with the actual program in the meta-variable $\$PGM$. The tables cell contains function, class and constant definitions. The scopes cell contains pointers to the various (global, super-global, current) scopes in the heap. The control cell contains the function stack, and information about the current object and class. The IO cell contains the input and output buffers, which $\mathbb{K}$ automatically connects to stdin and stdout. The instrumentation cell gathers meta-information for analysis purpose,

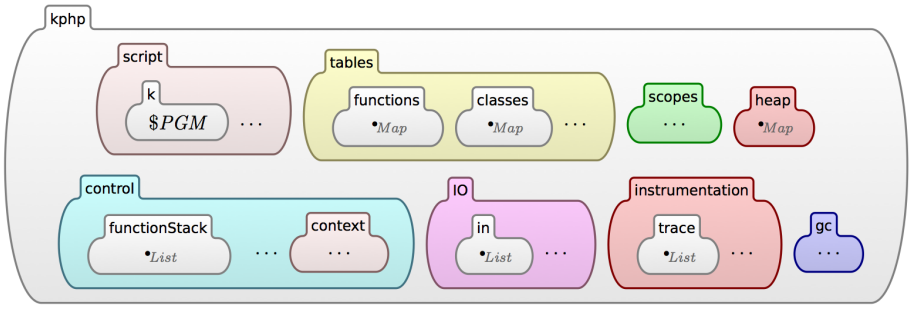

Fig. 2: Overview of the global configuration for $\mathbb{K}$PHP.

such as the trace of semantic rules used during an execution. Finally, the `gc` cell is used for bookkeeping by our implementation of garbage collection.

Semantic rules. Each non-trivial language construct is described by several rewrite rules, each performing a step towards the full evaluation of the construct. Conceptually, the evaluation process happens in three steps. First, *structural* and *context* rules are applied. Their role is to rearrange the current program so that other rules can be applied. These include for example *heating* and *cooling* rules, which move the arguments of an expression to the top of the computation stack (cell `k`) and plug the results back once evaluated, and *desugaring* rules. Next, *intermediate* rules apply. Their role is mostly to pre-process arguments. For example, they convert types, resolve references, or read from memory. Finally, *step* rules apply. They give semantics to the actual language constructs, and cause the term being evaluated to be consumed, returning a value where necessary, so that the computation may progress.

Besides rewrite rules, $\mathbb{K}$ definitions also include *functions*, which do not have side effects on the configuration. In our semantics, we mostly use these functions to define logical predicates for the side-conditions of other rewrite rules.

3.3 $\mathbb{K}$PHP: Selected Semantic Rules

Overall, the semantics comprises over 1,200 definitions: more than 700 are proper transition rules, the others are auxiliary definitions. As representative examples, we describe below the rules for assignment and functions.

Assignment. The rules for assignment are reported in Figure 3. Assignment is a binary expression whose arguments are evaluated left-to-right. We model this by two `CONTEXT` rules that enforce that evaluation order: rule (A) does not prescribe a type for the wildcard `_` variable, that can match any term, including an unevaluated expression, whereas rule (B) can apply only after the first argument is evaluated to a `KResult`. In an assignment, the LHS will evaluate to a reference. Rule (C) resolves the reference to a location. If the RHS is a value, rule (D) puts the internal operation `copyValueToLoc` at the front of the `k` cell, and puts the value `V` to be returned as a continuation. `copyValueToLoc` takes care of writing

(A) `CONTEXT 'Assign(□,_)`

(B) `CONTEXT 'Assign(_:KResult,□)`

(C) $\texttt{'Assign}\left(\frac{\texttt{R:Ref}}{\texttt{convertToLoc(R)}}, \_\right)$ `[intermediate]`

(D) $\frac{\texttt{'Assign(L:Loc, V:Value)}}{\texttt{copyValueToLoc(V, L)} \curvearrowright \texttt{V}}$ `[step]`

(E) $\texttt{'Assign}\left(\_ : \texttt{KResult}, \frac{\texttt{V:ConvertibleToLoc}}{\texttt{convertToLoc(V,r)}}\right)$
`when ¬ isLiteral(V) [intermediate]`

(F) $\frac{\texttt{'Assign(L:Loc,L1:Loc)}}{\texttt{reset(L)} \curvearrowright \texttt{'Assign(L, L1)}}$
`when currentOverflow(L1) [intermediate]`

(G) $\frac{\texttt{'Assign(L,L1)}}{\texttt{'Assign(L, convertToLanguageValue(L1))}}$
`when ¬ currentOverflow(L1) [intermediate]`

Fig. 3: Semantic rules for assignment.

`V` in `L`, operating recursively if `V` is an array. If the RHS is not a value, then rule (E) forces it to be converted to a location. If the location `L1` thus obtained contains an array whose current pointer is overflown, the current pointer of the assignment target `L` is reset by rule (F). In the remaining cases, rule (G) converts the location `L1` to a value, enabling rule (D).

Functions. When a function definition is executed, we create a new entry in the `functions` cell mapping the function name to a 4-tuple `f(FP,FB,RT,LS)` containing the function parameter list `FP`, its body `FB`, its return type `RT` (by value/by reference) and a pointer `LS` to a scope holding the static variables (which persist across function invocations).

A function call is parsed in the AST as `'FunctionCall(E,Args)`. Expression `E` is evaluated to a string `FN` (the function name), used to retrieve from the `functions` cell the various parameters described above. Execution continues by placing at the front of the `k` cell the internal `runFunction` term shown in the rule below, which replaces the contents of the cell `k` with a new list of internal commands (the current program continuation `K` will be saved on the stack).

$$\left\langle \frac{\texttt{runFunction(FN:String, f(FP:K, FB:K, RT:RetType, LS:Loc), Args:K)} \curvearrowright \texttt{K}}{\left(\begin{array}{r} \texttt{processFunArgs(FP, Args)} \curvearrowright \\ \texttt{pushStackFrame(FN, K, L, CurrentClass, CurrentObj, RT, D)} \curvearrowright \\ \texttt{ArrayCreateEmpty(L1)} \curvearrowright \texttt{setCrntScope(L1)} \curvearrowright \texttt{incRefCount(L1)} \curvearrowright \\ \texttt{copyFunArgs} \curvearrowright \texttt{FB} \curvearrowright \texttt{'Return(NULL)} \end{array}\right)} \right\rangle_{\texttt{k}}$$

$\langle \texttt{L:Loc} \rangle_{\texttt{currentScope}}$ $\langle \texttt{CurrentClass:Id} \rangle_{\texttt{class}}$ $\langle \texttt{CurrentObj:Loc} \rangle_{\texttt{object}}$

$\langle \frac{\texttt{D:K}}{\cdot} \rangle_{\texttt{functionArgumentDeclaration}}$

`when fresh(L1) [internal]`

The first command evaluates the function arguments `Args` in the current scope. This depends on the declaration of the formal parameters `FP`: if a parameter is declared by reference, the evaluation must stop at a location, and not fetch the value from memory. The next command pushes the current state on the stack, including the current scope `L` and continuation `K`. The next three commands create the function local scope `L1` (the side condition `fresh(L1:Loc)` means that location `L1` is newly allocated), set it as the current execution scope, and increment its reference counter. The next command assigns the evaluated arguments to the formal parameters allocated on `L1`. Finally, the function code `FB` is run, followed by a default `return` instruction.

3.4 The KPHP Interpreter

Our semantics is defined in 29 ".k" files, and consists of approximately 8500 lines of code. Compiling the semantics with the `kompile` utility of the K distribution creates a directory of files for the Maude tool. We provide a Unix shell script called `kphp` that, given the name of a PHP file, invokes the `krun` utility with appropriate parameters (for the external parser, options, etc.). This runs our semantics as if it was the standard PHP interpreter.

4 Testing and Validation

In this Section we discuss the testing and validation of our (executable) semantics of PHP. Since KPHP is actively developed, the numbers below refer to the release current at the time of publication of this paper.

Test-driven semantics development. As discussed in Section 1, there is no official document providing a specification of PHP. Hence, the development of our semantics was largely test-driven. The choice of specifying the semantics in K meant that at each stage of this work we had a working interpreter corresponding to the fragment of PHP we specified up to that point. This made it possible to test critical semantics rules as they were being developed. For this ongoing testing we wrote snippets of PHP, and compared the results from our interpreter with the ones from the Zend Engine, which is the *de facto* reference interpreter.

Validation. As common in other PHP projects (e.g. Facebook's HHVM), we validated our semantics/interpreter by testing it against the official test suite distributed with the Zend engine. Although our semantics covers most of the core PHP language (including its challenging features, such as arrays, objects, references, aliasing, exceptions, etc.), the test suite includes many tests that refer to constructs or library functions that we do not yet support. The Zend test suite is already split into folders containing different categories of tests. We tested the semantics against all the tests in the folders `lang` (core language) and `functions`: we did not pick tests manually to avoid introducing bias.

The `lang` folder contains 216 tests and we pass 97 of them, the `functions` folder contains 14 and we pass 4. If a test fails, it is for one of four reasons: (i) our semantics models a feature incorrectly; (ii) a language construct is not

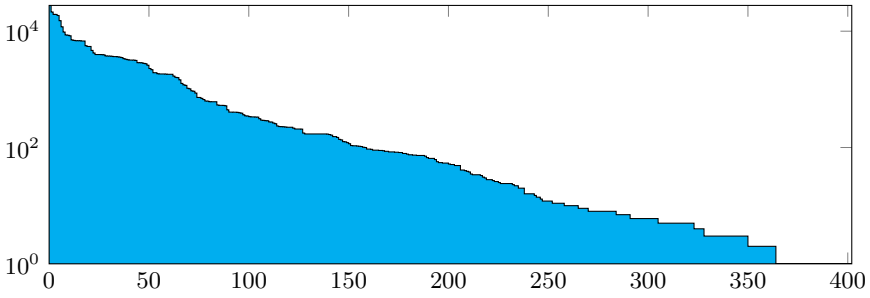

Fig. 4: Coverage of $\mathbb{K}$PHP rules by the Zend test suite (logarithmic scale).

supported by our semantics; (iii) the external parser has a bug, and returns the wrong AST; (iv) the external parser does not support some features added to PHP after version 5.0. For each failed test, our test harness shows one of these four categories. Successful tests are partitioned in 2 sets: 71 are automatically recognised as success, 26 are considered successful after manual review of the output (for example, our warning and error messages do not contain source code line numbers, because they are not recorded by the external parser).

The only test that fails in category (i) is `031.phpt`, which tests for the erratic behaviour of `current` described in Section 2.2. This fail is intentional, as we consider such behaviour to be a bug in the Zend Engine. All the other failed tests belong to categories (ii)-(iv), hence are either not supported by the semantics or by the parser. The total number of passed tests may not seem very high (JSCert passes almost 2,000 tests), but this is due to the size of the Zend test suite, and is outside our control. Moreover, many tests are non trivial, focussing on complex evaluation strategies, classes, constructors, interaction between global variables, functions, objects and aliasing. We are satisfied by this validation experiment - so far, our semantics behaves as the official PHP implementation.

Coverage. In order to assess the level of coverage of our semantics achieved by the Zend test suite, we added a `trace` cell to the $\mathbb{K}$PHP configuration, where we add the name of each rule as it is executed. Out of 721 semantics rules, 403 are executed at least once, and 318 are never executed. In Figure 4 we show the histogram, ordered by frequency, of the executed rules. There is a big difference in the number of times different rules are exercised by the test suite. This is partly explained by our design. The small group of rules which is called more than $25,000$ times by the test suite corresponds to the low-level, internal rules which are used as building blocks by other, higher level rules. Internal rules that perform type conversions, such as `*toInt`, are also intensively exercised, as expected. 158 of 403 rules are called at least 100 times, and 264 are called more than 10 times. Using the Zend tests alone, coverage amounts to 56% of the semantic rules. In order to achieve full-coverage, we have written targeted additional tests that cover the rules not exercised by the Zend suite.

5 Applications

One of the main goals of our semantics is to provide a formal model of PHP on which semantics-based verification tools (such as abstract interpreters, type systems and taint-checkers) can be built. Developing such tools goes beyond the scope of this paper. However, we are able to begin demonstrating the practical relevance of our semantics by showing potential applications based on the $\mathbb{K}$-Maude tool chain. In particular, the $\mathbb{K}$ framework exposes Maude's explicit-state LTL model checking to the semantics [11], and supports symbolic execution for any language definition [1]

In this Section, we show how we used LTL model checking in conjunction with symbolic execution to obtain a PHP code analyser, and we analyse properties of two 3rd-party PHP functions of practical relevance.

5.1 Temporal Verification of PHP Programs

Model checking and symbolic execution in $\mathbb{K}$. A $\mathbb{K}$ definition is eventually translated to a Maude rewrite theory, which can be model checked against LTL formulas using Maude's built-in model checker. In order to use the model checker in a meaningful way with respect to PHP, we need to instrument the semantics in two ways. First, we must decide what semantics rules should be considered as *state transitions* by the model checker, tag such rules, and pass the tags to the `--transition` compilation option. Second, we need to extend LTL with a set of atomic propositions that can be used to express interesting properties of PHP programs.

Symbolic execution has been recently introduced in $\mathbb{K}$ [1] and is enabled by using the option `--backend symbolic` when compiling the $\mathbb{K}$ definition. When symbolic mode is enabled, programs can be (optionally) given symbolic inputs of any of the types natively supported by the $\mathbb{K}$ tool (`int`,`float`,`string`,`bool`).

We now describe the $\mathbb{K}$PHP extensions needed for model checking and symbolic execution.

State transitions. Our semantics comprises many internal and intermediate rules, and it is not obvious *what* exactly should represent a change in the program state and what should instead be considered non-observable. Instead of fixing this notion once and for all, we allow ourselves maximum flexibility by defining several sets of semantic rules, and assigning a tag to each set:

- `step`: rules which correspond to the execution of language constructs.
- `internal`: rules used for operations which are not part of the user language, such as incrementing the reference counter.
- `intermediate`: rules which perform auxiliary work, such as performing a type conversion on an argument before a `step` or `internal` rule can be applied.
- `mem`: low-level rules which directly write the memory.
- `error`: orthogonal set of rules which cause a transition to an error state.

Using these tags, we are able to reason about programs at different degrees of abstraction. In the rest of this section, we consider only the state transitions generated by selecting the `step` rules. An alternative would be for example to consider only the `mem` rules, if the observations of interest are just the memory updates.

A temporal logic for PHP. We now define an extension of LTL with predicates over $\mathbb{K}$PHP configurations (i.e. PHP program states). Given a PHP program, we would like to be able to express conditions such as: "variable `$usr` never contains `'admin'`", or "the local variable `$y` of function `foo` will at some point be aliased to the global variable `$y`". Moreover, we also want to be able to reason about *correspondence assertions* [42], by labelling program points and stating properties such as "after reaching label `'login'` variable `$sec` always contains `1`". To this end, we introduce predicates such as `eqTo` (equals to), `gv` (global variable), `fv` (function variable). The LTL formulas corresponding to the informal specifications above are, respectively:

$\Box\neg$`eqTo(gv(var('usr')),val('admin'))`

$\Diamond$`alias(fv('foo',var('y')),gv(var('y')))`

`label('login')`$\Rightarrow\Box$`eqTo(gv(var(sec)),val(1))`

Moreover, we also found useful to be able to reason about types. For example, the following formula says that variable `$y` always has type `integer` during the execution of function `foo`:

$\Box($`inFun('foo')`$\Rightarrow$`has_type(fv('foo',var('y')),integer)`$)$

Each new predicate should be given a precise meaning in the context of the $\mathbb{K}$PHP configuration. We illustrate how we do that through the example of predicate `eqTo(e1,e2)`. Given a configuration `B`, we need to define when it satisfies the predicate:

`B` $\models$ `eqTo(e1,e2)` $\Leftrightarrow$ `eval(B,e1) = eval(B,e2)`

meaning, that formula `eqTo(e1,e2)` is true for `B` if and only if `e1` and `e2` evaluate to the same value. The definition of functions such as `eval` is crucial, as it connects the semantics to the model checker. These functions should be written in purely functional style and avoid side effects. In practice they are pretty simple, as they only need to inspect a configuration using pattern matching. As another example, consider predicate `alias(e1,e2)`:

`B` $\models$ `alias(e1,e2)` $\Leftrightarrow$ `lvalue(B,e1) = lvalue(B,e2)`

Function `lvalue` returns the location of the heap where its argument is stored, so the predicate is true when the arguments are aliased or identical. We use similar techniques to give semantics to all of our predicates, which can be used to form extended LTL formulas together with standard LTL connectives.

Limitations. The approach described here suffers from the known limitations of the underlying verification techniques. In particular, explicit state LTL model

checking will struggle to handle programs that generate large state spaces that depend heavily on the program inputs. The support for symbolic execution mitigates this problem, but as common to this approach it does not handle higher order data structures, such as objects, and struggles with loops depending on symbolic values. Despite these limitations, in the rest of this section we show that we can indeed verify some non-trivial properties of real PHP code.

5.2 Case Study: Input Validation

In our first example of model checking, we consider the function `PMA_isValid` taken from the source code of `phpMyAdmin` [33], one of the most common open source web applications, which provides a web interface to administer an SQL server.

`PMA_isValid` takes three arguments (`&$var`, `$type`, and `$compare`) and returns a boolean. Its purpose is to "validate" the argument `$var` according to different criteria that depend on the other two arguments. We analyse the full source code of `PMA_isValid`, which is shown in Appendix A.1 of the extended version of this paper, available on `http://phpsemantics.org`.

In the simplest case, `PMA_isValid` simply checks that `$var` is of the same type (or meta type) specified by `$type`, ignoring the remaining argument `$compare`:

```
PMA_isValid(0, "int");                  // true
PMA_isValid("hello", "scalar");         // true
PMA_isValid("hello", "numeric");        // false
PMA_isValid("123", "numeric");          // true
PMA_isValid("anything", false);         // always true
```

A more interesting case is when the argument `$type` is instantiated with one of `"identical"`, `"equal"` or `"similar"`. In such case, the validation of `$var` is performed against `$comparison`, according to the criterion specified by `$type`:

```
PMA_isValid(0, "identical", 1);         // false
PMA_isValid(0, "equal", 1);             // true
PMA_isValid("hello", "similar", 1);     // false
```

If `$type` is an array, validation succeeds if `$var` is an element of that array. If `$type` is `"length"`, validation succeeds if `$var` is a scalar with a string length greater that zero, If `$type = false`, validation always succeeds.

```
PMA_isValid(0, array(0,1,2));   // true
PMA_isValid(true, "length");    // true, as (string) true = "1"
PMA_isValid(false, "length");   // false, as (string) false = ""
```

The developer had an informal specification of this function in mind, which he wrote in a comment at the beginning of the function. However, it is not obvious wether such specification is met by the actual implementation. Leveraging model checking and symbolic execution we are able to prove that the function behaves as expected, by verifying each sub-case.

We first write some code accepting (possibly) symbolic inputs, and calling the function:

```
$var = user_input();              // symbolic
$type = user_input();             // symbolic
$compare = user_input();          // symbolic
$result = PMA_isValid($var, $type, $compare);
```

then we attempt to verify multiple times the LTL formula

$$\Diamond \texttt{eqTo(gv(var('result'),val(true)))}$$

each time providing different combinations of symbolic and concrete inputs, until all of the cases discussed above are covered. Indeed, these verifications succeed, proving the correctness of the function.

As a concrete example, in order to prove that the result of calling `PMA_isValid` with `$type="numeric"` is `true` when `$var` is an `integer`, we provide the symbolic input `#symInt(x)` to `$var`, and the concrete input `"numeric"` to `$type`. We proved analogous results for the case of `float` variables, and for the other similar cases. We proved that `PMA_is_Valid($var, "similar", $compare)` returns `true` for any `integer $var` and `string $compare`, by providing symbolic values `#symInt(x)` and `#symString(y)` to `$var` and `$compare`.

5.3 Case Study: Cryptographic Key Generation

In our second example, we consider the Password-Based Key Derivation Function `pbkdf2` from the PHP distribution [10]. `pbkdf2` takes five parameters: the name of the algorithm to be used for hashing (`$algo`), a `$password`, a `$salt`, an iteration `$count` and the desired `$key_length`. It returns a key derived from `$password` and `$salt` whose length is `$key_length`. We wish to prove that the function always returns a `string`, and that its length is equal to the requested `$key_length`.

Using the same approach as for the previous example, we write some initial code accepting (possibly) symbolic inputs, and calling the function:

```
$algo = "sha224";
$pass = user_input();   // symbolic input
$salt = user_input();   // symbolic input
$count = 1;
$key_len = 16;
$result = pbkdf2($algo, $pass, $salt, $count, $key_len);
```

Next, we run the model checker on our query formulae:

1. The result is a string: $\Diamond$`has_type(gv(var('result')),string)`
2. The length of the output is as requested:

$$\Diamond \texttt{eqTo(gv(var('key\_len')),len(gv(var('result'))))}$$

3. The length of the string stored in local variable `$output` grows, and eventually becomes greater than the required output length:

$$\Box\big(\ (\texttt{inFun('pbkdf2')} \wedge \neg \texttt{inFun('top')} \wedge \Diamond \texttt{inFun('top')}) \implies (\Diamond(\texttt{geq(len(fv('pbkdf2',var('output'))), fv('pbkdf2', var('key\_len')))}) \ \mathcal{U}\ \texttt{inFun('top')})\ \big)$$

Property (3) shows that property (2) is non-trivial. Moreover, it illustrates a technically more intriguing LTL formula. Using the model checker, we are able to verify that all three properties in less than a minute. Unlike the previous example, which we were able to run and analyse out-of-the-box, in this case we had to provide implementations for an number of functions (such as `hash`), which belong to libraries outside of the core language. For the sake of verification, we only provide simple stubs for these functions, making sure to preserve the type and output length properties of their original versions. The complete source code of `pbkdf2` and related functions can be found in Appendix A.2 of the extended version of this paper, available on `http://phpsemantics.org`.

6 Limitations and Future Work

A formal, executable semantics of a real programming language is too large a task to be completed in one single effort. This paper models the core of PHP, which includes the features we considered more important and instructive, leaving out some non-core features and the numerous language extensions. In this Section, we summarise what we left out of the current formalisation, and indicate what we think are the next priorities to take this work further.

Parsing limitations. As discussed in Section 4, the external parser we currently use does not understand some language constructs introduced after version 5.0, such as for example the literal array syntax with square brackets (`[1,2,3]` instead of `array(1,2,3)`). It also does not parse correctly some constructs such as `$this->a[]` which gets parsed as `$this->(a[])` instead of `($this->a)[]`. In future development, we plan to adopt a fully-compliant external parser.

Missing language features. We have not (yet) implemented a number of non-core language features, and in particular: bit-wise operators, most escape characters, regular expressions, namespaces, interfaces, abstract classes, iterators, magic methods and closures. We do not foresee significant obstacles in integrating these into $\mathbb{K}$PHP. For example, magic methods are special object methods (`__toString`, `__get`, `__call`, etc.) with reflective behaviour that are called automatically by PHP. JavaScript has similar reflective methods, and techniques to formalise them are well documented [25,4]. As a taster, we included in the core language the `__construct` magic method, which is used by the `new` command when creating a fresh object. Since version 5.3, PHP includes anonymous functions, implemented as objects of a special `Closure` class. These are not supported by our parser, but can be easily modelled in our semantics by adding to the object `OID` constructor an optional argument pointing to the entry of the `functions` cell where the anonymous function definition would be stored (using the same mechanism of regular functions).

Internal functions. As in related projects, a challenge when dealing with a real language is the sheer number of built-in library functions that operate on numbers, strings, arrays and objects. At the moment we model just a small, representative subset of them (e.g. `strlen`, `substr`, `count`, `is_int`, `is_float`, `var_dump`,

etc.). Where possible, we define such functions in PHP directly; we define them in $\mathbb{K}$ in the remaining cases (this corresponds to PHP native functions implemented in C).

Language extensions. Language extensions, such as the functions that provide access to an SQL database, or that connect a PHP script with a server (and hence with the network) are of fundamental importance for developing web applications, but are squarely beyond the scope of our current work. Our goal is to provide a sound semantic foundation to the core language that glues all such functions together. Until (semi-)automated techniques that help giving semantics to language extensions are developed, our view is that such extensions need to be investigated on a case-by-case basis. Often, they can be abstracted in terms of approximate information such as for example their types, taint behaviour, or side effects, as exemplified by our case study of Section 5.3.

7 Related Work

In this Section, we discuss related work on the mechanised formalisation of programming languages, and on the analysis of PHP.

Mechanised formalisation of programming languages. Proof assistants such as Coq [2], Isabelle/HOL [31], are a popular choice for the mechanised specification of programming languages. For example, HOL was used by Norrish [32] to specify a small-step operational semantics of C, and to prove meta-properties of the language. Blazy and Leroy, as part of the CompCert project [3], have built a verified compiler for a significant fragment of C, formalised in Coq. They have proved that the semantics of source programs is preserved by the compilation process. In the JSCert project [4], Bodin *et al.* have built a mechanised formalisation of JavaScript (ECMAScript 5) and tested it again the ECMA262 test suite. In Coq, they have developed an inductive definition of the semantics and a separate fixpoint definition of a JavaScript interpreter, and proved that the interpreter is sound with respect to the semantics. From the Coq fixpoint definition they have automatically extracted OCAML code to execute the interpreter.

In the $\mathbb{K}$ framework instead a semanticist may directly focus on writing and analysing language definitions: execution is taken care of by the tool. Ellison and Rosu [11] have defined an executable formal semantics of C in $\mathbb{K}$ [37]. Their formalisation has been extensively tested against the GCC torture suite [12], and they demonstrate examples of the debugging and model checking C code using the built-in capabilities of $\mathbb{K}$. On a smaller scale, $\mathbb{K}$, has also been used to formalise, Python [16], Scheme [28], Verilog [29], Haskell [23] and Java [5]. A number of program analysis techniques such as *symbolic execution* [1], *program logics* [36] and *program equivalence* [24] are being developed and incorporated into the $\mathbb{K}$, extending the potential benefits of defining a programming language semantics in this framework.

For PHP, we followed the approach of [11]. In the absence of a specification document, it was crucial to be able to immediately execute operational semantics rules as they were being developed, in order to compare with the reference

implementation of the language. Moreover, we were intrigued by the possibility to leverage existing model checking and symbolic execution capabilities to demonstrate our semantics at work.

The approach to define an executable semantics of a *real* programming language and validating it by testing against official test suites was trail-blazed by Guha *et al.* [15], who give semantics to JavaScript via a translation to a simpler intermediate language called λ_{JS}, formalised in the PLT Redex tool [22] (which also takes care of execution). More recently, the same approach was adopted by Politz *et al.* [35] to Python.

PHP analysis. Analysis of PHP and other web languages is an important topic, given the prevalence of security flaws such as XSS, CSRF and SQL injection. There are many research and commercial tools that statically analyse PHP code, including Pixy [21], WebSSARI [43], PHP-Sat [7] and HP Fortify [38]. According to the respective papers, all of these tools have specific weaknesses related to language features that are hard to understand and analyse. For example, Pixy and WebSSARI do not follow taint flows across objects. We believe that the next generation of static analysis tools can benefit from a precise, formal semantics of the language.

We are the first to present such a semantics for PHP. The only previous work we are aware of that looks in depth as some aspect of PHP semantics is an analysis of the array-copying mechanism of PHP by Tozawa *et al.* [41]. They formalise a tiny fragment of the language that suffices to describe the array copy mechanism, and show a flaw in a runtime optimisation used by the Zend engine. Their work sheds light on how complex the array semantics in PHP is, and was an inspiration for us to dig deeper into the PHP semantics.

8 Conclusions

In this paper we described the first formal semantics of PHP. We focussed on the core language, leaving language extensions and library functions for future work. Our semantics is executable, meaning that from the formal definition we automatically obtained a trusted interpreter of PHP, which we used for testing and debugging the semantics.

We validated the semantics by showing that is passes all the Zend tests applicable to the PHP fragment we modelled, and we achieved full coverage of our semantic rules by defining new ad-hoc tests. Given a mechanised semantics, it is still an open research problem how to automatically generate a comprehensive regression test suite for a language. If such a systematic approach to test generation was available, our semantics could be the basis for a regression test suite for PHP implementations.

Leveraging built-in features of $\mathbb{K}$ and Maude, we also provided a proof-of-concept infrastructure for the verification of PHP programs, which we demonstrated on two realistic examples. Our work is a first step towards defining semantics-based, static-analysis tools that provide formal guarantees for PHP web applications.

Acknowledgments. We are indebted to Antoine Delignat-Lavaud for many insightful discussions on the arcana of PHP. We would also like to thank the $\mathbb{K}$ team for their technical support on using the $\mathbb{K}$ framework, and Shijiao Yuwen for useful comments on an earlier version of the $\mathbb{K}$PHP semantics. Filaretti and Maffeis are supported by EPSRC grant EP/I004246/1.

References

1. A. Arusoaie, D. Lucanu, and V. Rusu. A Generic Framework for Symbolic Execution. In *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 281–301. Springer International Publishing, 2013.
2. Y. Bertot, P. Castéran, G. Huet, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development - Coq'Art - the Calculus of Inductive Constructions*. Texts in theoretical computer science. Springer, 2004.
3. S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43:263–288, 2009.
4. M. Bodin, , A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, A. Schmitt, and G. Smith. A Trusted Mechanised JavaScript Specification. In *POPL '14*, 2014.
5. D. Bogdanas. Formal Semantics of Java in the K Framework. `https://github.com/kframework/java-semantics`.
6. D. Bogdanas. Label-Based Programming Language Semantics in the K Framework with SDF. In *SYNASC'12*, pages 170–177, 2012.
7. E. Bouwers. PHP-Sat. `http://www.program-transformation.org/PHP/PhpSat`.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
9. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77*, pages 238–252, 1977.
10. PHP Documentation. Cryptographic Function pbkdf2. `http://php.net/manual/en/function.hash-hmac.php`.
11. C. Ellison and G. Roşu. An Executable Formal Semantics of C with Applications. In *POPL '12*, pages 533–544, 2012.
12. Free Software Foundation. GCC Torture Suite. `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.
13. The PHP Group. PHP Official Documentation. `http://www.php.net/manual/en/`.
14. The PHP Group. PHP Zend Engine. `http://php.net`.
15. A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of Javascript. In *ECOOP'10*, pages 126–150, 2010.
16. D. Guth. Python Semantics in K. `http://code.google.com/p/k-python-semantics/`.
17. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF - Reference Manual. *SIGPLAN Not.*, 24(11):43–75, 1989.
18. ECMA International. ECMA-262 ECMAScript Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, 2009.
19. International Organization for Standardization. C Language Specification - C11. ISO/IEC 9899:2011. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`, 2011.

20. R. Jhala and R. Majumdar. Software Model Checking. *ACM Compututing Surveys*, 41(4), 2009.
21. N. Jovanovic, C. Kruegel, and E. Kirda. Static Analysis for Detecting Taint-Style Vulnerabilities in Web Applications. *Journal of Computer Security*, 18(5):861–907, 2010.
22. C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, A. McCarthy, J, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *POPL '12*, pages 285–296, 2012.
23. D. Lazar. Haskell Semantics in K. `https://github.com/davidlazar/haskell-semantics`.
24. D. Lucanu and V. Rusu. Program Equivalence by Circular Reasoning. In *IFM'13*, pages 362–377, 2013.
25. S. Maffeis, J.C. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *APLAS '08*, pages 307–325, 2008.
26. S. Maffeis, J.C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS'2009*, pages 505–522, 2009.
27. S. Maffeis and A. Taly. Language-Based Isolation of Untrusted JavaScript. In *CSF'09*, pages 77–91, 2009.
28. P. Meredith, M. Hills, and G. Roşu. A K Definition of Scheme. Technical Report Department of Computer Science UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign, 2007.
29. P. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A Formal Executable Semantics of Verilog. In *MEMOCODE'10*, pages 179–188, 2010.
30. R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, 1997.
31. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002.
32. M. Norrish. C Formalised in HOL. University of Cambridge Technical Report UCAM-CL-TR-453, 1998.
33. phpMyAdmin Team. phpMyAdmin. `http://www.phpmyadmin.net/home_page/index.php`.
34. B.C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
35. J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the Full Monty. In *OOPSLA'13*, 2013.
36. G. Roşu and A. Ştefănescu. Checking Reachability using Matching Logic. In *OOPSLA'12*, pages 555–574, 2012.
37. G. Roşu and T.F. Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
38. Fortify Team. Fortify Code Secure. `http://www.armorize.com/codesecure/`.
39. K Team. The K Framework. `http://k-framework.org/index.php/Main_Page`.
40. PHP Quality Assurance Team. Zend Test Suite. `https://qa.php.net/write-test.php`.
41. A. Tozawa, M. Tatsubori, T. Onodera, and Y. Minamide. Copy-On-Write in the PHP Language. In *POPL'09*, pages 200–212, 2009.
42. T.Y.C. Woo and M. Lam. A Semantic Model for Authentication Protocols. In *Security and Privacy (SP)*, pages 178–194, 1993.
43. Y. Xie and A Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX'06*, 2006.