Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 331 (2017) 41–55

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Abstract Domains for Type Juggling

Vincenzo Arceri[1]

Department of Computer Science, University of Verona, Italy

Sergio Maffeis[2]

Department of Computing, Imperial College London, UK

Abstract

Web scripting languages, such as PHP and JavaScript, provide a wide range of dynamic features that make them both flexible and error-prone. In order to prevent bugs in web applications, there is a sore need for powerful static analysis tools. In this paper, we investigate how Abstract Interpretation may be leveraged to provide a precise value analysis providing rich typing information that can be a useful component for such tools.

In particular, we define the formal semantics for a core of PHP that illustrates *type juggling*, the implicit type conversions typical of PHP, and investigate the design of abstract domains and operations that, while still scalable, are expressive enough to cope with type juggling. We believe that our approach can also be applied to other languages with implicit type conversions.

Keywords: PHP, Static analysis, Abstract interpretation, Type conversions

1 Introduction

The success of web scripting languages such as PHP and JavaScript is also due to their wide range of dynamic features, which make them very flexible but unfortunately also error-prone. A key such feature is that language operations allow operands of any type, applying implicit type conversions when a specific type is needed. PHP, our example language, calls this feature *type juggling*.

In this paper, we investigate how the Abstract Interpretation approach to program analysis [3, 4] may be leveraged to provide a precise value analysis in presence of type juggling. Since PHP is dynamically typed, meaning that the same variable can store values of different types at different points in the execution, our analysis does not aim to enforce type invariance, but instead aims to determine the most precise type for each variable in the final state.

Filaretti and Maffeis [6] define a formal operational semantics for most of the PHP language that is faithful to its mainstream Zend reference implementation [1]. In Section 2, we propose $\mu\mathbb{PHP}$ (*micro*-PHP), a much smaller core of the language that is still large enough to illustrate the main challenges related to type juggling. In fact, $\mu\mathbb{PHP}$ *is* valid PHP, and behaves exactly like the full language[3], although the omission of certain language features from our formalisation (see Section 5) allows

[1] Email: vincenzo.arceri@studenti.univr.it

[2] Email: sergio.maffeis@imperial.ac.uk

[3] All the examples in the paper are both derivable via our semantics and executable in PHP 5.4.

http://dx.doi.org/10.1016/j.entcs.2017.02.003

us to define a more straightforward semantics than the one in [6]. We present $\mu\mathbb{PHP}$ in *big-step* semantics style, as we are interested in properties of the final state. [4] We show many examples that will reveal surprising behaviour of PHP to the non-expert.

In Section 3, we define an abstract semantics parametric on the domain, which defines a corresponding *flow-* and *path-sensitive* value analysis. We discuss assumptions on such domain under which we can argue that the analysis is sound with respect to the concrete semantics of $\mu\mathbb{PHP}$. The design of our semantics makes it straightforward to implement an abstract interpreter to calculate the analysis result.

In Section 4, we define abstract domains and operations that capture the subtleties of type juggling. Rather than giving the definitions upfront, we expound the rationale behind our design, stressing expressivity, modularity and hopefully highlighting subtle points that can be useful to design domains for other languages with similar features. Some practical static analyses of realistic languages with dynamic type conversions, such as [9, 11], add to each type lattice extra points that represent information which can improve the precision of the analysis. Other analyses, such as [8], use powersets of values, limiting the set sizes by a parameter k in order to avoid infinite computations. That leads to very expressive domains when up-to-k values are analysed, that drastically loose precision for further values.

In contrast, we advocate an expressive and systematic approach that refines each type domain to include just the information necessary to obtain precise abstract operations and type juggling functions. Our analysis may not be highly efficient but is scalable, having polynomial complexity: we emphasise precision over performance. As argued in [4], in theory one should aim for the *best correct approximation* of a concrete operator f defined as $f^\sharp = \alpha \circ f \circ \gamma$, but $f^\sharp$ is sometimes not computable, or practical. In defining the abstract operations of our type juggling domain we follow the spirit of this equation, striving to exploit at most the concrete information available, and delay as much as possible the loss of information caused by merging values with the $\sqcup$ operator.

Related Work. Since the seminal work of [2], abstract interpretation has been used to define many value and type analyses, but we are not aware of any analysis designed to handle in particular the implicit type conversions for scripting languages. On the practical side, several static analysers for JavaScript and PHP are directly based, or at least inspired, by abstract interpretation [5, 8–12]. All aim to analyse real-world PHP programs, and focus most effort on prominent issues such as the analysis of associative arrays and functions, while paying less attention to implicit type conversions. As far as we can tell (sometimes essential details are missing from the cited references), none of the analyses in [5, 9–12] comes close to our level of precision, except for [8] which, as discussed above, uses expensive powerset domains. Nevertheless, we hope that our investigation may contribute to improve the precision of these analysers for programs that make intensive use of implicit type conversions. Moreover, none of the cited works above provides formal proofs of soundness, and some such as [10, 12] openly admit to be unsound.

[4] It would be easy, but notationally more cumbersome, to define an equivalent *small-step* semantics better able to represent trace properties.

Summarising, our main claim of novelty is to apply a systematic approach grounded in the theory of Abstract Interpretation to analyse, in a provably sound way, non-trivial features of (the core of) a practical programming language.

2 Type Juggling in $\mu\mathbb{PHP}$

We now define syntax and semantics of $\mu\mathbb{PHP}$, a subset of PHP able to express most type juggling behaviour. Our examples can be verified in a PHP 5.x interpreter.

2.1 Syntax

To appreciate some subtle points of type juggling, we need to be somewhat precise about the representation of literals. Let $\textsc{Char}$ be the finite set of characters used in PHP, and $\textsc{Dig} \subsetneq \textsc{Char}$ the set of digits `0,...,9`. The literals of $\mu\mathbb{PHP}$ are partitioned in the sets

- $\textsc{Null}$: the constant `NULL`, which is the default value of undefined variables.
- $\textsc{Bool}$: the boolean constants `true` and `false`.
- $\textsc{Str} = \textsc{Char}^*$: strings such as `"hi!","","bye!"`.
- $\textsc{Int} = -^?\textsc{Dig}^+$: signed integers such as `-5,0,1,00042`.
- $\textsc{Float} = -^?\textsc{Dig}^*.\textsc{Dig}^*$: decimal notation numbers,[5] such as `-1.3,0.,4.200`.

The capitalisation of `NULL`, `true`, and `false` above is irrelevant. An empty sequence of digits between the optional sign and the decimal point of a float is interpreted as `0`, so for example `-.3` is an alternative representation for `-0.3`, and the degenerate case "." is not a valid $\textsc{Float}$. The syntax of $\mu\mathbb{PHP}$ is reported below:

$$\begin{array}{lcl}
\text{Lit} & ::= & \textsc{Null} \\
 & \mid & \textsc{Bool} \\
 & \mid & \textsc{Str} \\
 & \mid & \textsc{Int} \\
 & \mid & \textsc{Float} \\
\\
\text{Exp} & ::= & \text{Lit} \\
 & \mid & \text{Var} \\
 & \mid & ① \text{Exp} \\
 & \mid & \text{Exp} \; ② \; \text{Exp}
\end{array}
\qquad
\begin{array}{lcl}
\text{Var} & ::= & \texttt{\$}\textsc{Id} \\
\\
\text{Block} & ::= & \texttt{\{ \}} \\
 & \mid & \texttt{\{} \text{ Stmt } \texttt{\}} \\
\\
\text{Stmt} & ::= & \text{Var} \texttt{ = } \text{Exp} \texttt{ ;} \\
 & \mid & \texttt{if} \text{ (Exp) Block } \texttt{else} \text{ Block} \\
 & \mid & \texttt{while} \text{ (Exp) Block} \\
 & \mid & \text{Stmt Stmt} \\
 & \mid & \texttt{;}
\end{array}$$

where $\textsc{Id}$ is a subset of $\textsc{Str}$ suitable to define identifiers. We denote prefix unary operators by ① $\in$ {`!`,`-`,`+`} and infix binary operators by ② $\in$ {`+`,`-`,`*`,`/`,`%`,`&&`,`||`,`==`,`!=`,`>`,`<`,`>=`,`<=`}.

[5] PHP floats normally use the IEEE 754 double precision format. For simplicity, we use instead decimal numbers in $\mu\mathbb{PHP}$.

2.2 Semantics

Semantic values correspond to literals, but abstract away from representation details. In particular, leading zeros are dropped when parsing an Int, except for the literal `0`, and leading and trailing zeros are dropped when parsing a Float, so `-004.20` is the semantic float `-4.2`. With a slight abuse of notation, we use the same font to denote literal and values, as the meaning should be clear from the context. Str, Int, and Float are finite sets, and floating point numbers have limited precision. We denote by Num the union Int∪Float, and by Val the union of all the semantic values above. For any set S and $X \subseteq S$, we also define the notation $\overline{X}$ for the complement of X with respect to S.

Program states State : Id $\longrightarrow$ Val, ranged over by σ, are partial functions from identifiers to values. State updates and lookups are defined as follows:

$$\sigma[\mathtt{x} \hookleftarrow v](\mathtt{y}) = \begin{cases} v & \text{if } \mathtt{x} = \mathtt{y} \\ \sigma(\mathtt{y}) & \text{otherwise} \end{cases}$$

Statements. The big-step semantics of blocks and statements is defined by the function $[\![\cdot]\!]\cdot$: Stmt × State $\longrightarrow$ State defined below

$$\begin{aligned}
[\![\mathtt{\$x = e;}]\!]\sigma &= \sigma[\mathtt{x} \hookleftarrow [\![\mathtt{e}]\!]\sigma] \\
[\![\mathtt{if\ (e)\ bl1\ else\ bl2}]\!]\sigma &= \begin{cases} [\![\mathtt{bl1}]\!]\sigma & \text{if } \mathsf{toBool}([\![\mathtt{e}]\!]\sigma) = \mathtt{true} \\ [\![\mathtt{bl2}]\!]\sigma & \text{if } \mathsf{toBool}([\![\mathtt{e}]\!]\sigma) = \mathtt{false} \end{cases} \\
[\![\mathtt{while\ (e)\ bl}]\!]\sigma &= [\![\mathtt{if\ (e)\ \{\ bl\ while\ e\ bl\ \}\ else\ \{\ \}}]\!]\sigma \\
[\![\mathtt{\{\ S\ \}}]\!]\sigma &= [\![\mathtt{S}]\!]\sigma \\
[\![\mathtt{\{\ \}}]\!]\sigma &= [\![\mathtt{;}]\!]\sigma = \sigma \\
[\![\mathtt{S1\ S2}]\!]\sigma &= [\![\mathtt{S2}]\!]([\![\mathtt{S1}]\!]\sigma)
\end{aligned}$$

All the rules are standard except for the if-else, which contains the first example of type juggling, where the value resulting from evaluating the guard expression `e` in state σ is then automatically converted to a boolean, using the function toBool defined below, where $\text{Num}_0 = \{\mathtt{0},\mathtt{0.0}\}$, $\text{Str}_{\mathtt{false}} = \{\mathtt{""},\mathtt{"0"}\}$.

$$\mathsf{toBool}(v) = \begin{cases} v & \text{if } v \in \text{Bool} \\ \mathtt{false} & \text{if } v \in \text{Null} \cup \text{Num}_0 \cup \text{Str}_{\mathtt{false}} \\ \mathtt{true} & \text{if } v \in \overline{\text{Num}_0} \cup \overline{\text{Str}_{\mathtt{false}}} \end{cases}$$

This leads us to our first example of odd behaviour in PHP:

```
php> if (0) {echo "yes";} else {echo "no";}      // "no"
php> if ("0") {echo "yes";} else {echo "no";}    // "no"
php> if (0.0) {echo "yes";} else {echo "no";}    // "no"
php> if ("0.0") {echo "yes";} else {echo "no";} // "yes"
```

Expressions. The semantics of expressions is given by the function $[\![\cdot]\!]\cdot$: Exp × State $\longrightarrow$ Val which we describe case-by-case below. The semantics of a literal is just the corresponding parsed value, as described at the beginning of this Section. The variable rule returns the value of the corresponding identifier, if it is defined in the current state, and NULL otherwise.

$$[\![\texttt{\$x}]\!]\sigma = \begin{cases} \sigma(\texttt{x}) & \text{if } \texttt{x} \in dom(\sigma) \\ \texttt{NULL} & \text{otherwise} \end{cases}$$

Arithmetic operations are defined on any type of operands:

$$[\![\texttt{e1} ② \texttt{e2}]\!]\sigma = \mathsf{toNum}([\![\texttt{e1}]\!]\sigma)\ \boxed{2}\ \mathsf{toNum}([\![\texttt{e2}]\!]\sigma)$$

where the operands are converted to numbers (integers or floats) via another type juggling function toNum. Let $\mathsf{parseNum} : \textsc{Str} \longrightarrow (\textsc{Num} + \{\perp\}) * \textsc{Str}$ be a function that returns the number that can be parsed as the largest prefix of a string (if any), and the remainder of the string that does not contribute to parsing the number. For example, $\mathsf{parseNum}(\texttt{".42000.37hi"}) = (\texttt{0.42}, \texttt{".37hi"})$ and $\mathsf{parseNum}(\texttt{"bye666"}) = (\perp, \texttt{"bye666"})$. The function toNum is defined by

$$\mathsf{toNum}(v) = \begin{cases} v & \text{if } v \in \textsc{Int} \cup \textsc{Float} \\ \texttt{1} & \text{if } v = \texttt{True} \\ \texttt{0} & \text{if } v \in \textsc{Null} \cup \{\texttt{False}\} \\ \texttt{0} & \text{if } \mathsf{parseNum}(v) = (\perp, v) \\ n & \text{if } \mathsf{parseNum}(v) = (n, s) \text{ for some } s \end{cases}$$

When ② $\in \{\texttt{+},\texttt{-},\texttt{*}\}$, $\boxed{2}$ corresponds to the most precise corresponding primitive operation between integers and floats (denoted by $\{+, -, *\}$). So, for example:

```
php> var_dump(3.2*"hi" + 45 - "3bye"*true); // float(42)
```

When ② $\in \{\texttt{/},\texttt{\%}\}$ instead, $\boxed{2}$ implements a μℙℍℙ-specific function that returns `false` when division by zero occurs.

$$n_1 \boxed{/} n_2 = \begin{cases} n_1/n_2 & \text{if } n_2 \in \overline{\textsc{Num}_0} \\ \texttt{false} & \text{if } n_2 \in \textsc{Num}_0 \end{cases}$$

The semantics of comparison operators is tricky, as it depends on the type of the operands. For example, to compare a string with a boolean, first it is converted to a boolean, and then both booleans are compared after being converted to numbers, leading to the perhaps surprising example below.

```
php> var_dump("0" < true);   // bool(true)
php> var_dump("0.0" < true); // bool(false)
```

More formally, we define the semantics for the less-than operator as follows (the other comparison operators follow a similar pattern):

$$[\![\texttt{e1 < e2}]\!]\sigma = [\![\texttt{e1}]\!]\sigma\ \boxed{<}\ [\![\texttt{e2}]\!]\sigma$$

$\boxed{<}$	Int	Float	Bool
Int	$v_1 < v_2$	$v_1 < v_2$	$\mathsf{toNum}(\mathsf{toBool}(v1)) < \mathsf{toNum}(v2)$
Float	$v_1 < v_2$	$v_1 < v_2$	$\mathsf{toNum}(\mathsf{toBool}(v1)) < \mathsf{toNum}(v2)$
Bool	$\mathsf{toNum}(v_1) < \mathsf{toNum}(\mathsf{toBool}(v_2))$	$\mathsf{toNum}(v_1) < \mathsf{toNum}(\mathsf{toBool}(v_2))$	$\mathsf{toNum}(v_1) < \mathsf{toNum}(v_2)$
Str	$\mathsf{toNum}(v_1) < v_2$	$\mathsf{toNum}(v_1) < v_2$	$\mathsf{toNum}(\mathsf{toBool}(v_1)) < \mathsf{toBool}(v_2)$
Null	$\mathsf{toNum}(v_1) < v_2$	$\mathsf{toNum}(v_1) < v_2$	$\mathsf{toNum}(v_1) < \mathsf{toNum}(v_2)$

$\boxed{<}$	Str	Null
Int	$v_1 < \mathsf{toNum}(v_2)$	$v_1 < \mathsf{toNum}(v_2)$
Float	$v_1 < \mathsf{toNum}(v_2)$	$v_1 < \mathsf{toNum}(v_2)$
Bool	$\mathsf{toNum}(v_1) < \mathsf{toNum}(\mathsf{toBool}(v_2))$	$\mathsf{toNum}(v_1) < \mathsf{toNum}(v_2)$
Str	$\mathsf{toNum}(v_1) < v_2$	$\mathsf{toNum}(\mathsf{toBool}(v_1)) < \mathsf{toBool}(v_2)$
Null	$\mathsf{toStr}(v_1) <_{\text{STR}} v_2$	`false`

Figure 1. Tables with semantics rules for the less-than operator applied to basic values

When `e1` and `e2` reach final values v_1 and v_2, the semantics rules reported in Figure 1 are applied, where $<$ is the primitive operator of less-than for numbers, and $<_{\text{STR}}$ is a non-standard comparison between strings. If two strings can be parsed exactly as numbers, they are compared using $<$ on the parsed numbers; otherwise, they are compared in the lexicographic order $<_L$.

$$s_1 <_S s_2 = \begin{cases} n_1 < n_2 & \text{if } \mathsf{parseNum}(s_1) = (n_1, \texttt{""}) \text{ and } \mathsf{parseNum}(s_2) = (n_2, \texttt{""}) \\ s_1 <_L s_2 & \text{otherwise} \end{cases}$$

This leads to more surprising behaviour. For example,

```
php> var_dump("10"<"9");           // bool(false)
php> var_dump("10LOW"<"9HIGH");    // bool(true)
php> var_dump(0+"10LOW"<"9HIGH");  // bool(false)
```

where the use of + in the third example forces the use of toNum on the first string (hence on the second one too), and the use of $<$ instead of $<_{\text{STR}}$ in the comparison.

The semantics of string concatenation is defined as follows

$$[\![\texttt{e1.e2}]\!]\sigma = \mathsf{toStr}([\![\texttt{e1}]\!]\sigma) \boxdot \mathsf{toStr}([\![\texttt{e2}]\!]\sigma)$$

where $\boxdot$ is the primitive operation of string concatenation. The type juggling function toStr is defined below, where $\text{FLOAT}_{\text{INT}} = \text{INT}^{?}\texttt{.0}^{*}$ (excluding the degenerate case ".") represents the floats that can be interpreted as integers without approximation, such as `.00`, `42.`, `0.0`. When an element of $\text{FLOAT}_{\text{INT}}$ is concatenation with a string, only its integer part is concatenated.

$$\mathsf{toStr}(v) = \begin{cases} \texttt{"1"} & \text{if } v = \texttt{true} \\ \texttt{""} & \text{if } v = \texttt{false} \\ \texttt{"}v\texttt{"} & \text{if } v \in \textsc{Int} \cup \overline{\textsc{Float}_{\textsc{Int}}} \\ \texttt{"}u\texttt{"} & \text{if } v \in \textsc{Float}_{\textsc{Int}} \text{ and } u = \mathsf{floor}(v) \\ v & \text{if } v \in \textsc{Str} \end{cases}$$

Above, $\mathsf{floor} : \textsc{Num} \longrightarrow \textsc{Int}$ rounds down its argument to the nearest integer.

3 Abstract Interpretation of $\mu\mathbb{PHP}$

Our goal is to design an efficient value analysis that retains precise information on the type of variables. Hence, our *concrete domain* representing the properties of interest is the standard complete lattice $\langle 2^{\textsc{Val}}, \subseteq \rangle$. With the above goal in mind, we now define an abstract semantics for $\mu\mathbb{PHP}$ that is parametric in the choice of an abstract domain of values $\langle \textsc{Val}^\sharp, \sqsubseteq \rangle$.

3.1 Abstract Semantics

Our analysis is non-relational, hence we can somewhat simplify the design of the abstract semantics and the definition of its soundness properties. In particular, abstract program states $\textsc{State}^\sharp : \textsc{Id} \rightarrow \textsc{Val}^\sharp$, ranged over by ξ, can partition the available information *per identifier*, and be defined as partial functions from identifiers to abstract values. State updates and lookups are defined as for the concrete semantics.

Statements. The abstract semantics of blocks and statements $[\![\cdot]\!]^\sharp\cdot : \text{Stmt} \times \textsc{State}^\sharp \longrightarrow \textsc{State}^\sharp$ is similar to the concrete one.

$$\begin{aligned} [\![\texttt{\$x = e;}]\!]^\sharp\xi &= \xi[x \hookleftarrow [\![\texttt{e}]\!]^\sharp\xi] \\ [\![\texttt{\{ S \}}]\!]^\sharp\xi &= [\![\texttt{S}]\!]^\sharp\xi \\ [\![\texttt{\{ \}}]\!]^\sharp\xi &= [\![;]\!]^\sharp\xi = \xi \\ [\![\texttt{S1 S2}]\!]^\sharp\xi &= [\![\texttt{S2}]\!]^\sharp([\![\texttt{S1}]\!]^\sharp\xi) \end{aligned}$$

The rules for assignment, blocks and sequences are analogous to the ones for the concrete semantics. Note that in particular we are considering *strong* updates to the state: our analysis is *flow-sensitive.*

$$[\![\texttt{if (e) bl1 else bl2}]\!]^\sharp\xi = \begin{cases} [\![\texttt{bl1}]\!]^\sharp\xi & \text{if } \gamma(\mathsf{toBool}^\sharp([\![\texttt{e}]\!]^\sharp\xi)) = \{\texttt{true}\} \\ [\![\texttt{bl2}]\!]^\sharp\xi & \text{if } \gamma(\mathsf{toBool}^\sharp([\![\texttt{e}]\!]^\sharp\xi)) = \{\texttt{false}\} \\ [\![\texttt{bl1}]\!]^\sharp\xi \sqcup [\![\texttt{bl2}]\!]^\sharp\xi & \text{if } \gamma(\mathsf{toBool}^\sharp([\![\texttt{e}]\!]^\sharp\xi)) \supseteq \textsc{Bool} \end{cases}$$

The rule for if-else is *path-sensitive*, mimicking the concrete one, yet includes a conservative extra case when the evaluation of the guard does not result in a precise boolean value. It relies on an abstract type juggling function $\mathsf{toBool}^\sharp$ which is to be defined together with the abstract domain $\langle \textsc{Val}^\sharp, \sqsubseteq \rangle$, as discussed in Section 4.2.

$$[\![\texttt{while (e) bl}]\!]^\sharp \xi = \mathit{lfp}_\xi(\lambda \rho.(\rho \sqcup [\![\texttt{if (e) bl else \{\}}]\!]^\sharp \rho))$$

The rule for while loops in the concrete semantics can be equivalently formulated as $[\![\texttt{while (e) bl}]\!]\sigma = \mathit{lfp}_\sigma([\![\texttt{if (e) then bl else \{\}}]\!])$: the abstract rule is simply a conservative approximation, whose computability depends on the definition of abstract domain.[6]

Expressions. The abstract evaluation of expressions is denoted by $[\![\cdot]\!]^\sharp \cdot : \text{Exp} \times \textsc{State}^\sharp \longrightarrow \textsc{Val}^\sharp$. Literal values are simply abstracted by the rule $[\![\text{Lit}]\!]^\sharp \xi = \alpha(\text{Lit})$. The abstract variable look-up rule is analogous to the concrete one, except that looking up an undefined identifier returns $\alpha(\texttt{NULL})$ instead of `NULL`. Depending on the choice of $\textsc{Val}^\sharp$ and the definition of α, that could be a specific element $\texttt{NULL}^\sharp$, or $\top$, or a different abstract element.

$$[\![\texttt{\$x}]\!]^\sharp \xi = \begin{cases} \xi(\texttt{x}) & \text{if } \texttt{x} \in \mathit{dom}(\xi) \\ \alpha(\texttt{NULL}) & \text{otherwise} \end{cases}$$

The abstract evaluation of arithmetic expressions is analogous to the concrete case

$$[\![\mathsf{e1} \ \textcircled{2}\ \mathsf{e2}]\!]^\sharp \xi = \mathsf{toNum}^\sharp([\![\mathsf{e1}]\!]^\sharp \xi) \ \boxed{2}^\sharp \ \mathsf{toNum}^\sharp([\![\mathsf{e2}]\!]^\sharp \xi)$$

where $\boxed{2}^\sharp$ is the abstract operation corresponding to $\boxed{2}$, and $\mathsf{toNum}^\sharp$ is the abstract type juggling function corresponding to toNum. Both $\boxed{2}^\sharp$ and $\mathsf{toNum}^\sharp$ are to be defined along with the abstract domain on which they depend. The abstract semantics of the other expressions follows a similar pattern.

3.2 Soundness of the analysis

We argue that the class of analyses defined by our abstract semantics is sound, assuming that the abstract domain has the right structure, and that the abstract operations provided with such domain satisfy some local soundness conditions.

Assumption 3.1 (Abstract Domain) *The abstract domain $\langle \textsc{Val}^\sharp, \sqsubseteq \rangle$ is a complete lattice, and it forms a Galois connection $2^{\textsc{Val}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \textsc{Val}^\sharp$ with the concrete domain $\langle 2^{\textsc{Val}}, \subseteq \rangle$.*

Assumption 3.2 (Abstract Operations) *The abstract operations provided with the domain $\langle \textsc{Val}^\sharp, \sqsubseteq \rangle$ are* monotonic *and* locally sound *approximations of the concrete ones: $\forall f^\sharp. \forall u, v \in \textsc{Val}^\sharp : u \sqsubseteq v \Rightarrow f^\sharp(u) \sqsubseteq f^\sharp(v)$ and $\forall f, f^\sharp. \forall v \in \textsc{Val} : \alpha(f(v)) \sqsubseteq f^\sharp(\alpha(v))$.*

We can take advantage of the big-step style of our semantics, and of our interest in properties of the final state, to bypass the standard definition of a collecting

[6] Our abstract semantics does not use boolean filter functions, because it is not practical to define realistic ones for a programming language as complicated as PHP. This choice has the downside of sacrificing some precision in the semantics of while loops, because we do not refine the information in the abstract state at the end of the loop to reflect that the guard has to be false .

semantics and state our soundness theorem directly in terms of the concrete and abstract semantics. We only need to lift the definition of α from values to states: $\alpha(\sigma) = \alpha \circ \sigma$, and similarly for $\gamma, \sqsubseteq$.

Theorem 3.3 (Soundess) *The abstract semantics is a sound approximation of the concrete semantics:* $\forall s \in Stmt : \alpha \circ [\![s]\!] \sqsubseteq [\![s]\!]^\sharp \circ \alpha$.

Proof By induction on the derivation of $[\![\cdot]\!]\cdot$ (joining the definition for statements and expressions), using Assumption 3.1, Assumption 3.2 and standard properties of lattices. We show the case for if-else which is representative of the other cases. Assume that $\mathsf{toBool}([\![\mathsf{e}]\!])\sigma = \texttt{true}$ (the case when $\mathsf{toBool}([\![\mathsf{e}]\!]) = \texttt{false}$ is analogous). Let $\xi = \alpha(\sigma)$. By inductive hypothesis, $\alpha([\![\texttt{bl1}]\!]\sigma) \sqsubseteq [\![\texttt{bl1}]\!]^\sharp\xi$. By definition, $[\![\texttt{bl1}]\!]^\sharp\xi \sqsubseteq [\![\texttt{bl1}]\!]^\sharp\xi \sqcup [\![\texttt{bl2}]\!]^\sharp\xi$. Hence, we only need to exclude the case where $\gamma(\mathsf{toBool}^\sharp([\![\mathsf{e}]\!]^\sharp\xi)) = \{\texttt{false}\}$. By Assumption 3.2, $\alpha(\mathsf{toBool}([\![\mathsf{e}]\!]\sigma)) \sqsubseteq \mathsf{toBool}^\sharp(\alpha([\![\mathsf{e}]\!]\sigma))$. By inductive hypothesis, $\alpha([\![\mathsf{e}]\!]\sigma) \sqsubseteq [\![\mathsf{e}]\!]^\sharp\xi$. By monotonicity of $\mathsf{toBool}^\sharp$, $\mathsf{toBool}^\sharp(\alpha([\![\mathsf{e}]\!]\sigma)) \sqsubseteq \mathsf{toBool}^\sharp([\![\mathsf{e}]\!]^\sharp\xi)$. By transitivity of $\sqsubseteq$, $\alpha(\mathsf{toBool}([\![\mathsf{e}]\!]\sigma)) \sqsubseteq \mathsf{toBool}^\sharp([\![\mathsf{e}]\!]^\sharp\xi)$. By assumption, $\mathsf{toBool}([\![\mathsf{e}]\!])\sigma = \texttt{true}$. If $\gamma(\mathsf{toBool}^\sharp([\![\mathsf{e}]\!]^\sharp\xi)) = \{\texttt{false}\}$, substituting in the equations above, we obtain $\gamma(\alpha(\texttt{true})) \subseteq \{\texttt{false}\}$. By Assumption 3.1, $\{\texttt{true}\} \subseteq \gamma(\alpha(\texttt{true}))$, which leads to the contradiction $\{\texttt{true}\} \subseteq \{\texttt{false}\}$. $\square$

Proposition 3.4 (Incompleteness) *The abstract semantics is not complete:*

$$\exists s.\ \alpha \circ [\![s]\!] \subsetneq [\![s]\!]^\sharp \circ \alpha.$$

Proof We show that there is a counterexample even for the most precise abstract domain possible: $\langle 2^{\text{VAL}}, \subseteq\rangle$ itself, where α and γ are the identify function. Let `P` be the $\mu\mathbb{PHP}$ program `$x=1; while ($x>0){ $x=$x-1; }`. For any $\sigma \in \textsc{State}$, we have

$$(\alpha \circ [\![\texttt{P}]\!])(\sigma) = \sigma[\texttt{x} \hookleftarrow \{0\}] \subsetneq \sigma[\texttt{x} \hookleftarrow \{0,1\}] = ([\![\texttt{P}]\!]^\sharp \circ \alpha)(\sigma). \qquad \square$$

The informal meaning of our formal results is that if our analysis finds that a certain property holds, then that property (or possibly a stronger one) also holds across all the concrete executions compatible with the initial abstract state.

4 Abstract Domains for Type Juggling

Equipped with the abstract semantics of Section 3, we can design abstract domains and operations that capture the subtlety of type juggling in $\mu\mathbb{PHP}$. Rather than giving the definitions upfront, we expound the rationale behind our design, stressing expressivity, modularity and hopefully highlighting subtle points that can be useful to design domains for other languages with similar features.

4.1 Abstract Domains

We face three main design choices: how to combine the abstraction of the various types of $\mu\mathbb{PHP}$; how to abstract each type; how to ensure that we can represent as much of the information relevant to type juggling as possible.

Type combination. Let us assume that for each set of basic values T we have defined an abstract type lattice $T^\sharp$. A typical analysis for statically-typed languages may combine abstract types using the *coalesced sum* lattice, which in our case yields

$$\begin{tikzcd}
 & & & \top \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
\textsc{Val}^\sharp = & \textsc{Null}^\sharp & \textsc{Bool}^\sharp & \textsc{Int}^\sharp & \textsc{Float}^\sharp & \textsc{Str}^\sharp \\
 & & & \bot \arrow[ull, dash] \arrow[ul, dash] \arrow[u, dash] \arrow[ur, dash] \arrow[urr, dash] & &
\end{tikzcd}$$

This choice is not appropriate for a dynamically-typed language such as $\mu\mathbb{PHP}$, as the resulting lattice cannot represent union types. For example, in the lattice above it must be the case that $\alpha(5) \sqcup \alpha(3.2) = \top$, leading to an unnecessary loss of precision when we convert such value to a string, because it has to be the case that $\mathsf{toStr}^\sharp(\top) = \top$. In contrast, in a domain with the union type $\textsc{Int}^\sharp + \textsc{Float}^\sharp$, $\mathsf{toStr}^\sharp$ could have retained the information that numbers are never converted to empty strings, allowing to derive $\mathsf{toStr}^\sharp(\textsc{Int}^\sharp + \textsc{Float}^\sharp) = \textsc{Str}^\sharp_{\neq \texttt{""}}$, assuming that the type abstraction of strings was able to account for such elements. A common solution to this problem consists in switching to the *cartesian product* lattice of the abstract types

$$\textsc{Val}^\sharp = \textsc{Null}^\sharp \times \textsc{Bool}^\sharp \times \textsc{Int}^\sharp \times \textsc{Float}^\sharp \times \textsc{Str}^\sharp$$

where, for example, the union type $\textsc{Int}^\sharp + \textsc{Float}^\sharp$ is implicitly represented by the vector $(\bot, \bot, \textsc{Int}^\sharp, \textsc{Float}^\sharp, \bot)$.[7]

Type abstraction. Another key design choice is how to abstract the types themselves. Fore example, consider the $\mu\mathbb{PHP}$ semantics of division. It normally returns a $\textsc{Num}$ except for the case of division-by-zero, where it returns `false`. Abstracting a value directly to its type, as in $\alpha(5) = \textsc{Int}^\sharp$, is too imprecise because it prevents an analysis from detecting the division-by-zero case, and it forces the return type to be at best $\textsc{Num}^\sharp + \textsc{Bool}^\sharp$, instead of the more precise $\textsc{Num}^\sharp + \texttt{false}^\sharp$. Hence, we include also the constants of each type to the product lattice. We define $\textsc{Null}^\sharp$ as the *lift*, and $\textsc{Bool}^\sharp$ and $\textsc{Str}^\sharp$ as the *flat* lattices built from the corresponding sets:

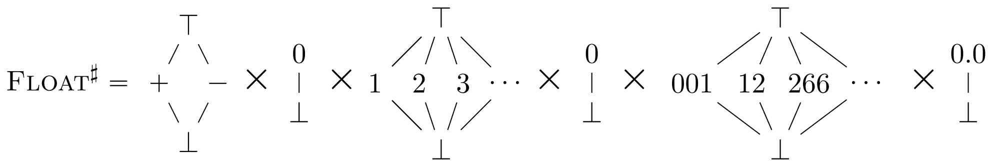
$$\textsc{Null}^\sharp = \mathsf{lift}(\textsc{Null}) \qquad \textsc{Bool}^\sharp = \mathsf{flat}(\textsc{Bool}) \qquad \textsc{Str}^\sharp = \mathsf{flat}(\textsc{Str})$$

By a judicious definition of $\textsc{Int}^\sharp$ as the product lattice of signs, the constant 0, and natural numbers, we obtain the discriminating power of the traditional sign domain, plus the precision of numeric constants.

$$\textsc{Int}^\sharp = \mathsf{flat}(\{+, -\}) \times \mathsf{lift}(\{0\}) \times \mathsf{flat}(\mathbb{N})$$

[7] The definition of α and γ will be left implicit as it can be understood from the context, as the obvious best approximation.

For example, $(\top, \bot, \top)$ denotes non-zero integers, and $(+, 0, \top)$ represents non-negative integers.[8] A similar argument applies to $\textsc{Float}^\sharp$, which we define as



and is isomorphic to $\textsc{Int}^\sharp \times \mathsf{lift}(\{0\}) \times \mathsf{flat}(\textsc{Frac}) \times \mathsf{lift}(\{0.0\})$, where $\textsc{Frac}$ is the set of non-zero "fractional parts" denoted by the regular expression [0..9]*[1..9]. The last component of the product is necessary to distinguish $\alpha(\texttt{0.0}) \sqcup \alpha(\texttt{1.2})$, which can be zero, from $\alpha(\texttt{0.1}) \sqcup \alpha(\texttt{1.0})$, which cannot. For notational convenience, we denote the abstraction $n^\sharp$ within the type domain $T^\sharp$ by $\alpha_{T^\sharp}(n)$. We also abbreviate the bottom element of a product type, such as $(\bot, \bot, \bot) : \textsc{Int}^\sharp$ simply by $\bot$ (and similar for $\top$). Finally, we use the shorthand $\top_{\textsc{Bool}^\sharp}$ for the element $(\bot, \top, \bot, \bot, \bot) : \textsc{Val}^\sharp$, with the obvious generalisation to other elements or domains.

Type juggling. Thanks to the definitions above, most of our domains already include enough information to handle type juggling. For example, the definition of toBool depends on the set $\textsc{Num}_0$=$\{\texttt{0,0.0}\}$. In order to define a precise abstract $\mathsf{toBool}^\sharp$, we should avoid loss of precision when deciding if an abstract value, once concretised, belongs to $\textsc{Num}_0$. Our domain achieves that, because for example $\gamma(\alpha(\texttt{0}) \sqcup \alpha(\texttt{0.0})) = \textsc{Num}_0$, and similarly $\gamma(\alpha(\texttt{5}) \sqcup \alpha(\texttt{-3.2})) = \overline{\textsc{Num}_0}$. The only domain which we need to refine explicitly is that of strings. In fact, toBool also relies on the set $\textsc{Str}_{\texttt{false}} = \{\texttt{"","0"}\}$, but if $\textsc{Str}^\sharp$ is just the flat string domain, then $\gamma(\alpha(\textsc{Str}_{\texttt{false}})) = \textsc{Str} \neq \textsc{Str}_{\texttt{false}}$. A solution to this specific problem is to add to $\textsc{Str}^\sharp$ elements representing exactly $\alpha(\textsc{Str}_{\texttt{false}})$ and $\alpha(\overline{\textsc{Str}_{\texttt{false}}})$. The downside is that repeating this process for the other operations leads to a proliferation of special cases. For example, the division operation needs to decide if the result of toNum is in $\textsc{Num}_0$. Hence, for a precise $\mathsf{toNum}^\sharp$ we need two new points in $\textsc{Str}^\sharp$ representing precisely $\alpha(\{\texttt{"0","0.0"}\})$ and its complement. Moreover, we would need to introduce additional structure in the lattice to compare these points and the ones representing $\alpha(\textsc{Str}_{\texttt{false}})$, $\alpha(\overline{\textsc{Str}_{\texttt{false}}})$, and so on. Our proposal is instead to simply add all the information that is missing from the $\textsc{Str}^\sharp$ domain by adding to strings additional properties reflecting their value *after* an hypothetical type juggling. We re-define $\textsc{Str}^\sharp$ as a product involving also booleans, integers and floats, interpreted as properties of the corresponding abstract string:

$$\textsc{Str}^\sharp = \mathsf{flat}(\textsc{Str}) \times \textsc{Bool}^\sharp \times \textsc{Int}^\sharp \times \textsc{Float}^\sharp$$

All the points representing properties of interest hypothesised above now are included in the lattice, with the correct ordering relation. For example, the

[8] Some points in our lattice, such as $(+, 0, \bot)$ are redundant (zero has no sign). It is possible to optimise the domains to remove such points, slightly increasing the efficiency of the analysis (although the precision remains the same). We leave investigating that direction to future work.

string type of $\alpha(\text{STR}_{\texttt{false}})$ is $(\top, \texttt{false}, \texttt{0}^\sharp, \bot)$, whereas the one of $\alpha(\texttt{"0"}, \texttt{"0.0"})$ is $(\top, \top, \texttt{0}^\sharp, \texttt{0.0}^\sharp)$. As a final example of the expressivity of our type juggling domain $\text{VAL}^\sharp$, let x be the abstract value $\alpha(\texttt{"0.0doh"}) \sqcup \alpha(\texttt{42})$, which in our domain is $(\bot, \bot, \texttt{42}^\sharp, \bot, (\texttt{"0.0doh"}, \texttt{true}, \bot, \texttt{0.0}^\sharp))$. Our domain contains enough information to be able to infer that x is not `NULL`, that it is `true` if converted to a boolean, and that the abstract evaluation of `84/x` yields $(\bot, \texttt{false}^\sharp, \texttt{2}^\sharp, \bot, \bot)$, assuming a suitable definition of $\boxed{/}^\sharp$ (see Section 4.2).

4.2 Abstract Operations

We now discuss how to implement abstract operations that take advantage of the information represented by $\text{VAL}^\sharp$.

Type juggling functions. We focus on the example of $\mathsf{toNum}^\sharp$ as it illustrates all the main issues at hand. Since an abstract value is actually a 5-tuple of individual abstract types, in order to retain precision, we convert each component independently, using specialised functions such as $\mathsf{StrToNum}^\sharp : \text{STR}^\sharp \rightarrow \text{VAL}^\sharp$, where the result is either an abstract number or $\bot$. Hence, the type of $\mathsf{toNum}^\sharp$ is $\text{VAL}^\sharp \longrightarrow (\text{VAL}^\sharp)^5$. Note that we do not collapse the resulting 5-tuple into a single $\text{VAL}^\sharp$ so that the operation that invoked the type juggling operation can leverage the information at best. For example,

$$\mathsf{toNum}^\sharp((\bot, \bot, 4^\sharp, \bot, \texttt{"6doh"}^\sharp)) = (\bot_{\text{VAL}^\sharp}, \bot_{\text{VAL}^\sharp}, 4_{\text{INT}^\sharp}, \bot_{\text{VAL}^\sharp}, 6_{\text{INT}^\sharp})$$

and a division by $2_{\text{INT}^\sharp}$ can return "positive integer" instead of "positive number". The specialised conversions $\mathsf{StrToNum}^\sharp$, $\mathsf{BoolToNum}^\sharp$, etc. are straightforward to define, following their concrete counterparts. For example, the latter returns respectively $\bot_{\text{VAL}^\sharp}, 0_{\text{INT}^\sharp}, 1_{\text{INT}^\sharp}, (+, 0, 1)$ on the inputs $\bot, \texttt{true}^\sharp, \texttt{false}^\sharp, \top$. Without loss of precision, we define $\mathsf{StrToNum}^\sharp$ as the function $\lambda x.\pi_3(x) \sqcup \pi_4(x)$ that joins the pre-computed conversions to integer and float associated to the $\text{STR}^\sharp$ value.

Semantic operations. We now discuss how abstract operations can leverage the expressiveness of our domain. We give the example of division, which is representative of the other cases. Since $\mathsf{toNum}^\sharp$ has already been applied by the abstract semantics of expressions, we now have to divide two 5-tuples of $\text{VAL}^\sharp$, hence $\boxed{/}^\sharp : {\text{VAL}^\sharp}^5 \times {\text{VAL}^\sharp}^5 \longrightarrow \text{VAL}^\sharp$.

The first step of $\boxed{/}^\sharp$ is to *normalise* each tuple by removing any $\bot_{\text{VAL}^\sharp}$ value, and retaining only its numeric components greater than $\bot$, obtaining two vectors of at most 6 elements each. For example, let $v = \mathsf{toNum}^\sharp(\alpha(\texttt{true}) \sqcup \alpha(\texttt{"-5foo"}) \sqcup \alpha(\texttt{"4.2doh"})) = (\bot_{\text{VAL}^\sharp}, 1_{\text{INT}^\sharp}, \bot_{\text{VAL}^\sharp}, \bot_{\text{VAL}^\sharp}, \alpha(\texttt{-5}) \sqcup \alpha(\texttt{4.2}))$. By normalising, we obtain $\mathsf{n}(v) = [\texttt{1}^\sharp, \texttt{-5}^\sharp, \texttt{4.2}^\sharp]$.

Once we have two normalised (row) vectors z and w, we can compute the analogous of the matrix product $z^t \times 1/w$, effectively obtaining a matrix r of dimension $|z| \times |w|$ where $r_{i,j} = z[i] /^\sharp w[j]$, and $/^\sharp : (\text{INT}^\sharp + \text{FLOAT}^\sharp)^2 \longrightarrow \text{VAL}^\sharp$ is the abstract division operator defined below

$$n_1/^\sharp n_2 = \begin{cases} \alpha(m_1 \boxed{/} m_2) & \text{if } \gamma(n_1) = \{m_1\} \text{ and } \gamma(n_2) = \{m_2\} \\ n_1/^\sharp_{\text{INT}^\sharp} n_2 & \text{else, if } n_1, n_2 \text{ are both INT}^\sharp \\ \mathsf{toFloat}^\sharp(n_1)/^\sharp_{\text{FLOAT}^\sharp}\mathsf{toFloat}^\sharp(n_2) & \text{otherwise} \end{cases}$$

where $\mathsf{toFloat}^\sharp : \text{INT}^\sharp \to \text{FLOAT}^\sharp$ maps $\alpha_{\text{INT}^\sharp}(k)$ to $\alpha_{\text{FLOAT}^\sharp}(k.0)$. The final step of $\boxed{/}^\sharp$ is to join all the elements of r into a single $\text{VAL}^\sharp$. We define

$$u \boxed{/}^\sharp v = \bigsqcup_{\substack{i\in 1..|x| \\ j \in 1..|y|}} x[i]/^\sharp y[j] \qquad \text{where } x = \mathsf{n}(u) \text{ and } y = \mathsf{n}(v).$$

The abstract division operator $/^\sharp$ relies on specialised abstract divisions for integers and floats (respectively $/^\sharp_{\text{INT}^\sharp}$ and $/^\sharp_{\text{FLOAT}^\sharp}$). When both operands are abstract integers, we perform a further normalisation, separating the information about 0 from the information about $\mathbb{N}$ encoded in each operand. For example, $\mathsf{n}((-, 0, 5)) = [(-, \bot, 5), (\bot, 0, \bot)]$. Then, we compute $u/^\sharp_{\text{INT}^\sharp} v = \bigsqcup_{\substack{i\in 1..|x| \\ j \in 1..|y|}} x[i]/^\sharp_{\text{INT}^\sharp}/y[j]$ where $x = \mathsf{n}(u)$ and $y = \mathsf{n}(v)$, and the inner $/^\sharp_{\text{INT}^\sharp}$ is computed using the rules in Figure 2. The case for $/^\sharp_{\text{FLOAT}^\sharp}$ is analogous.

For example, let us revisit the example of `84/x` from Section 4.1, where this time

$$x = \alpha(\texttt{"0.0doh"}) \sqcup \alpha(\texttt{"1argh"}) \sqcup \alpha(\texttt{42}) \sqcup \alpha(\texttt{0}) = (\bot, \bot, (+, 0, 42), \bot, (\top, \texttt{true}^\sharp, \texttt{1}^\sharp, \texttt{0.0}^\sharp))$$

We have that $\mathsf{n}(\mathsf{toNum}^\sharp(x)) = [\texttt{1}^\sharp, \texttt{0.0}^\sharp, (+, 0, 42)]$. The first two divisions are computed directly as $\alpha(\texttt{84} \boxed{/} \texttt{1}) = \alpha(\texttt{84})$ and $\alpha(\texttt{84} \boxed{/} \texttt{0.0}) = \alpha(\texttt{false})$. The third division is computed as $\texttt{84}^\sharp/^\sharp_{\text{INT}^\sharp}(+, 0, 42)$. The denominator is normalised to $[\texttt{0}^\sharp, \texttt{42}^\sharp]$, leading to two further divisions $\alpha(\texttt{84} \boxed{/} \texttt{0}) = \alpha(\texttt{false})$ and $\alpha(\texttt{84} \boxed{/} \texttt{42}) = \alpha(\texttt{2})$. Hence, the final result is $\alpha(\texttt{84}) \sqcup \alpha(\texttt{false}) \sqcup \alpha(\texttt{2}) = (\bot, \texttt{false}^\sharp, (+, \bot, \top), \top, \top)$, where we know that, unless there was a division by zero, we obtain a positive integer. Note that both normalisation steps introduced by $\boxed{/}^\sharp$ and $/^\sharp_{\text{INT}^\sharp}$ were essential to retain this level of precision.

5 Conclusions

We have defined the formal semantics of $\mu\mathbb{PHP}$, a subset of PHP that precisely represents type juggling behaviour, as a basis to explore new and expressive abstract domains for type/value analysis. We have also defined an abstract interpreter that implements, parametrically on the domain, a non-relational, path-sensitive analysis to leverage our abstract domains. We have shown with various examples that our value analysis is more expressive than comparable ones present in the literature. To the best of our knowledge, a novelty of our approach is the definition of the string domain as the product of the string type with other abstract types (integers and floats). This construction helps retaining more precise information about strings after type juggling.

$/^\sharp_{\textsc{Int}^\sharp}$	$0^\sharp$	$1^\sharp$	$n_2^\sharp$	$(\top, \bot, n_2)$
$0^\sharp$	$\texttt{false}^\sharp$	$0^\sharp$	$0^\sharp$	$0^\sharp$
$1^\sharp$	$\texttt{false}^\sharp$	$1^\sharp$	$\alpha(1 \boxed{/} n_2)$	$\alpha(1 \boxed{/} n_2) \sqcup \alpha(1 \boxed{/} -n_2)$
$n_1^\sharp$	$\texttt{false}^\sharp$	$n_1^\sharp$	$\alpha(n_1 \boxed{/} n_2)$	$\alpha(n_1 \boxed{/} n_2) \sqcup \alpha(n_1 \boxed{/} -n_2)$
$(\top, \bot, n_1)$	$\texttt{false}^\sharp$	$(\top, \bot, n_1)$	$\alpha(-n_1 \boxed{/} n_2) \sqcup \alpha(n_1 \boxed{/} n_2)$	$\alpha(n_1 \boxed{/} n_2) \sqcup \alpha(-n_1 \boxed{/} n_2) \sqcup \alpha(n_1 \boxed{/} -n_2) \sqcup \alpha(-n_1 \boxed{/} -n_2)$
$(+, \bot, \top)$	$\texttt{false}^\sharp$	$(+, \bot, \top)_{\textsc{Int}^\sharp}$	$(\pi_1(n_2^\sharp), \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\pi_1(n_2^\sharp), \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(-, \bot, \top)$	$\texttt{false}^\sharp$	$(-, \bot, \top)_{\textsc{Int}^\sharp}$	$(\overline{\pi_1(n_2^\sharp)}, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\overline{\pi_1(n_2^\sharp)}, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(\top, \bot, \top)$	$\texttt{false}^\sharp$	$(\top, \bot, \top)_{\textsc{Int}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(\top, 0, \top)$	$\texttt{false}^\sharp$	$(\top, 0, \top)_{\textsc{Int}^\sharp}$	$(\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$

$/^\sharp_{\textsc{Int}^\sharp}$	$(+, \bot, \top)$	$(-, \bot, \top)$	$(\top, \bot, \top)$	$(\top, 0, \top)$
$0^\sharp$	$0^\sharp$	$0^\sharp$	$0^\sharp$	$\texttt{false}^\sharp \sqcup 0^\sharp$
$1^\sharp$	$(+, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (+, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(-, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (-, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(+, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (+, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$\texttt{false}^\sharp \sqcup (\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$n_1^\sharp$	$(\pi_1(n_1^\sharp), \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\pi_1(n_1^\sharp), \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\overline{\pi_1(n_1^\sharp)}, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\overline{\pi_1(n_1^\sharp)}, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$\texttt{false}^\sharp \sqcup (\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(\top, \bot, n_1)$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$\texttt{false}^\sharp \sqcup (\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(+, \bot, \top)$	$(+, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (+, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(-, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (-, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$\texttt{false}^\sharp \sqcup (\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(-, \bot, \top)$	$(-, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (-, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(+, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (+, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$\texttt{false}^\sharp \sqcup (\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(\top, \bot, \top)$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, \bot, \top)_{\textsc{Int}^\sharp} \sqcup (\top, \bot, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$\texttt{false}^\sharp \sqcup (\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$
$(\top, 0, \top)$	$(\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$(\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$	$\texttt{false}^\sharp \sqcup (\top, 0, \top)_{\textsc{Int}^\sharp} \sqcup (\top, 0, \top, 0, \top, \bot)_{\textsc{Float}^\sharp}$

Figure 2. Tables for the abstract operation $/^\sharp_{\textsc{Int}^\sharp}$.

The main limitations of our current work also suggest natural directions for future work. $\mu\mathbb{PHP}$ covers only a small subset of PHP, and it will be interesting to see how our type juggling domain interacts with the analyses of other challenging language features such as aliasing, functions, objects and exceptions. Our construction of the type juggling domain strives to be systematic but we do not investigate how an analysis of completeness of the abstract operations, along the lines of [7], may lend further justification to our current design choices, or lead to the completely automated construction of a more precise domain.

Acknowledgments. This work is partially supported by EPSRC grant EP/K032089/1.

References

[1] The PHP Group. PHP Zend Engine. http://php.net. Accessed: 2016-06-09.

[2] P. Cousot. Types as abstract interpretations. In *POPL'97*, 1997.

[3] 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*, 1977.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79*, 1979.

[5] J. Dahse and T. Holz. Simulation of built-in PHP features for precise static code analysis. In *NDSS'14*, 2014.

[6] D. Filaretti and S. Maffeis. An executable formal semantics of PHP. In *ECOOP'14*, 2014.

[7] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 2000.

[8] D. Hauzar and J. Kofron. Framework for static analysis of PHP applications. In *ECOOP'15*, 2015.

[9] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *SAS'09*, 2009.

[10] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *(S&P'06)*, 2006.

[11] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: a static analysis platform for javascript. In *FSE'14*, 2014.

[12] E. Kneuss, P. Suter, and V. Kuncak. Phantm: PHP analyzer for type mismatch. In *FSE'10*, 2010.