Object-sensitive Type Analysis of PHP

Henk Erik van der Hoek Jurriaan Hage

Department of Computing and Information Sciences
Utrecht University
mail@henkerikvanderhoek.nl,J.Hage@@uu.nl

Abstract

In this paper we develop an object-sensitive type analysis for PHP, based on an extension of the notion of monotone frameworks to deal with the dynamic aspects of PHP, and following the framework of Smaragdakis et al. for object-sensitive analysis.

We consider a number of instantiations of the framework to see how the choices affect the running cost of the analysis, and the precision of the outcome. In this setting we have not been able to reproduce the major gains reported by Smaragdakis et al., but do find that abstract garbage collection substantially increases the scalability of our analyses.

Categories and Subject Descriptors D.3.2 [*Software*]: Language Classifications—Object-oriented languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.3.4 [*Programming Languages*]: Processors – *soft typing, PHP*

General Terms Languages, Theory, Verification

Keywords static analysis, monotone frameworks, PHP, object-sensitivity, abstract garbage collection, cost and precision

1. Introduction

Statically typed languages perform type checking at compile time, allowing type errors to be caught at the earliest possible stage, and enabling compilers to perform optimizations. However, since the type checker runs at compile time it must be conservative and reject programs that may execute correctly. Also, programming in a language with explicit type checking is sometimes perceived as more difficult.

In dynamically typed languages type checking is performed at run-time. This implies that type errors are only caught at run-time, which means that programs only fail if a type error does show up. Moreover, in many dynamic languages, apparent type inconsistencies are often resolved by silently coercing values to another type. This may easily lead to actual errors remaining undetected. Moreover, to perform run-time type checking, every value must be tagged in some way by a description of its type, which leads to substantial overheads. For example, type analysis plays an important role in reducing run-time overhead in HipHop, a PHP compiler developed by Facebook [23], and has later led to the Hack language that seamlessly integrates with PHP and adds type annotations [22]. Moreover, type information can improve features like on-the-fly auto-completion, and type related error detection.

```
 1 class Value {
 2   function evaluate () {
 3     $v = $this->v;
 4     return $v;
 5   }
 6 }
 7
 8 class Multiply {
 9   function evaluate () {
10     $l = $this->l;
11     $x = $l->evaluate ();
12     $r = $this->r;
13     $y = $r->evaluate ();
14     $z = $x * $y;
15     return $z;
16   }
17 }
18 $x = new Value ();
19 $v = 10;
20 $x->v = $v;
21 $y = new Value ();
22 $v = false;
23 $y->v = $v;
24 $z = new Multiply ();
25 $z->l = $x;
26 $z->r = $y;
27 $r = $z->evaluate ();
```

Figure 1. A lowered expression evaluator in PHP

Researchers have tried to overcome the problems of dynamically typed languages and gain the advantages of static typing by performing a form of (soft) type inference at compile time, including [3, 4, 6, 8, 10, 19]. Type analysis determines *for each program point, which types each variable may have at the exit of that point*. To illustrate the analysis, consider the simple expression evaluator in Figure 1 after it has been lowered to the core syntax that our analysis supports. The type analysis should for example determine that the variable $\$l$ on line 10 is an object of type `Value` and that the variable $\$x$ on line 11 is an integer while variable $\$y$ on line 13 is a Boolean. Note that in order to determine the receiver method of the method call on line 11, line 13 and line 27, the type of the receiver object has to be known. This behaviour, known as dynamic method dispatching, results in a mutual dependency between the control flow and the propagated type information.

Type analysis for dynamic languages is ofted phrased as a dataflow analysis for which standard solutions such as monotone frameworks exist [16]; much of what we do is based on this stan-

PEPM'15, January 13-14 2015, Mumbai, India.
Copyright © 2015 ACM 978-1-4503-3297-2/15/01…$15.00.
http://dx.doi.org/10.1145/2678015.2682535

dard work on static analyses. Often, context-sensitivity is employed to increase precision, in the form of call-site sensitivity: the analysis results for different calls are separated with the help of call strings or contours that describe an abstraction of the call stack that led to the given program point. In [15], a different approach to context is described that uses the creation points of objects instead of the labels of call strings; this is called object sensitivity. We use the framework of [20] as our starting point, to investigate how well different instances of their framework do within the context of type analysis for PHP. In this paper, we

- specify an object-sensitive type analysis for PHP as an instance of a monotone framework that supports dynamic control flow edges discovery (see Section 4),
- specify several analysis variations of the analysis (see Section 5), and
- implement a prototype of the type analysis, and report on an experimental evaluation of its precision and performance for different analysis variations (see Section 6).

2. Preliminaries

To pave the way for later sections, we shortly discuss the variant of monotone frameworks that we employed in our work. PHP is a dynamic language that supports a number of features (higher-order functions, dynamically added methods (to objects and classes), and subtyping) for which pre-computing a static call graph is not reasonable. For this reason we employ a variation of the embellished monotone frameworks as described in [16]. It allows us to discover new control flow edges on-the-fly, comparable to [10] and [8]. Intuitively, it discovers new flow edges in the same way as implementations of embellished monotone frameworks discover sensible context values (for a given program point). For reasons of space we omit a detailed discussion of this aspect of our work, and refer instead to [21] for a further discussion. We only reiterate here what is necessary to understand what follows.

A particular choice we made in our formulation of monotone frameworks is what is exactly an "item of work" to be stored in the worklist algorithm that is employed by our variant of the maximal fixed point algorithm [16, Chapter 2]. In our case such an item describes that execution can reach a particular program point under a given context: $(l, \delta) \in \textbf{Point} = \textbf{Label} \times \textbf{Context}$.

The language PHP

When dealing with an actual programming language, it typically does not pay to deal with the complete syntax of the language, but only those aspects that are of interest, for example because other language constructs can be desugared into these; this is also the approach we take here. This is not problematic since we are primarily interested in the sets of types computed for the variables in the program, not in providing feedback about these sets to the programmer. A pleasant side effect is that it also allows us to phrase our analysis more concisely. We do note that there are aspects of PHP that our analysis does not cover: string coercion (by calling `__toString`), object cloning (using `clone`), namespaces, treating strings as arrays of characters, anonymous functions, references, `eval`, include files with non-literal file names, object destructors, and parts of the Standard PHP Library (in particular, `ArrayAccess`, `Clonable` and `Iterator`).

3. Control Flow Graphs for PHP

We shall specify the type analysis over an intermediate representation (IR). The intermediate representation captures the key operations which are necessary to perform an object sensitive typing analysis. For example, various looping constructs are rewritten to an equivalent `while` loop and composite expressions, e.g., $a = b * 2 + c$; are lowered to a sequence of simple statements, $tmp1 = b*2; a = tmp1 + c$. Furthermore, since class and function identifiers are case insensitive in PHP all class and function identifiers are rewritten to their lowercase form. In addition to rewriting, the translation to the intermediate representation also takes care of including files. Note that these simplifications do not change the expressiveness of the language. A subset of the IR that we call core IR will be formally specified below. It includes features like class definitions, object allocations, method invocations and field reads and writes. A formal description of our type analysis on the core IR is given in Section 4. For additional constructs such as native functions, exceptions and arrays we refer to [21].

$$
\begin{array}{lll}
\mathrm{P} & ::= & \mathrm{C\ S} \\
\mathrm{C} & ::= & \textbf{class}\ \mathrm{c\ M} \mid \textbf{class}\ \mathrm{c}\ \textbf{extends}\ \mathrm{c\ M} \mid \mathrm{C\ C} \\
\mathrm{M} & ::= & [\textbf{function}\ \mathrm{m}\ (\vec{p})]^{l_n}_{l_x}\ \mathrm{S} \mid \mathrm{M\ M} \\
\mathrm{S} & ::= & [\mathrm{v} = \mathrm{L}]^l \mid [\mathrm{v} = \mathrm{v}]^l \mid [\mathrm{v} = \mathrm{v\ op\ v}]^l \mid \mid \mathrm{S}_1\ \textbf{;}\ \mathrm{S}_2 \mid [\textbf{skip}]^l \\
 & \mid & \textbf{if}\ [\mathrm{v}]^l\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2 \\
 & \mid & \textbf{while (true)}\ \mathrm{S} \mid [\textbf{break}]^l \mid [\textbf{continue}]^l \\
 & \mid & [\textbf{new}\ \mathrm{c}]^l \mid [\mathrm{v.f} = \mathrm{v}]^l \mid [\mathrm{v} = \mathrm{v.f}]^l \mid [\mathrm{v} = \mathrm{v.m}\ (\vec{p})]^{l_c}_{l_r} \\
 & \mid & [\textbf{return}\ \mathrm{v}]^l \\
\mathrm{L} & ::= & \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid \mathrm{n}
\end{array}
$$

Figure 2. The syntax of the core language of PHP

We assume the following syntactic categories:

n	$\in$ **Num**	Integers
x, y	$\in$ **Var**	Variables
c	$\in$ **ClassName**	Class names
f	$\in$ **FieldName**	Field names
m	$\in$ **MethodName**	Method names
op	$\in$ **Operators**	Binary operators

A syntax for IR Core is given in Figure 2, describing the syntax of programs (P), classes (C), statements (S) and literals (L) respectively. We adopt terminology from [16], e.g., everything between square brackets is called an *elementary block*. We assume each elementary block is annotated with a distinct label, e.g., l.

We write **Method** for **ClassName** $\times$ **MethodName**, Figure 3 contains a number of auxiliary functions to be used by our analysis (following [16], when we write, e.g., $init_*$ to refer to $init$ for the progam under analysis).

4. The type analysis

Type analysis determines *for each program point, which types each variable may have at the exit of that point.* To compute an approximation of the types for variables in a given program, we first lower a PHP application to the syntax given in the previous section. We can then perform a type analysis, in tandem with a *points-to* analysis. A points-to analysis computes a static approximation of all the heap objects that a pointer variable can refer to at run-time. PHP lacks a syntactic difference between pointer variables (which point to heap allocated objects) and regular variables. Not knowing whether a variable may be a pointer variable our type analysis computes a static approximation of all the values that a variable may point to at run-time. This approach is similar to the approach taken in [10] to perform type analysis of JavaScript programs.

Type analysis and control flow information are mutually dependent: suppose class `A` and `B` both support (different) methods called `foo`, and we can establish by means of our type analysis that the variable `x` may be of type `A`, but not `B`. Then we can conclude that the call to `x.foo()` can only pass control to the `foo` of `A` and not

$init$: **Program** $\rightarrow \mathcal{P}$(**Label**) returns the initial labels of the program.
$flow$: **Program** $\rightarrow$ **Flow** returns the intraprocedural flow graph; interprocedural edges will be added on-the-fly during fixed point iteration.
$entry$: **Program** $\times$ **Method** $\rightarrow$ **Label** returns the exit label l_n of a method.
$exit$: **Program** $\times$ **Method** $\rightarrow$ **Label** returns the exit label l_x of a method.
$return$: **Program** $\times$ **Label** $\rightarrow$ **Label** returns the return label l_r corresponding to a given call label l_c.
$resolve$: **Program** $\times$ **Method** $\rightarrow$ **Method** resolves a dynamic method call by traversing the inheritance hierarchy, until the method looked for is encountered.
$className$: **Program** $\times$ **Label** $\rightarrow$ **ClassName** returns the class name of the allocated object given an allocation site label.

Figure 3. Auxiliary functions for retrieving information about programs

the `foo` of `B`. Since the body of `B.foo` is not analyzed at this point, this may in turn lead to smaller points-to/type sets.

To deal with the dynamic method dispatching of PHP (that it shares with other object-oriented languages), also the flow of abstract state information and call graph information are mutually dependent. This is why we employ a variation of Embellished Monotone Frameworks to deal with this dynamic aspect of the language (as shortly discussed in Section 2).

4.1 The type lattice

We model our abstract value as a 4-tuple. Each component of the tuple contains an abstraction for a specific type: integers, booleans, objects, and null values. One can therefore view our type analysis as an extension of a points-to analysis since the tuple component for objects contains the points-to information.

The abstract representation of sets of integer values is by means of sign sets, $\mathcal{P}$(**Sign**) where **Sign** $= \{\texttt{-}, \texttt{0}, \texttt{+}\}$, which form a complete lattice under set inclusion. The function

$$fromInteger : \mathbf{N} \rightarrow \mathbf{Sign}$$

abstracts in the usual way integers to signs. In our implementation we provide abstract operators like $\hat{+}$ to compute precisely with abstract values.

The abstract Booleans are elements of the set $\mathcal{P}$(**Bool**) where **Bool** $= \{\texttt{true}, \texttt{false}\}$. In this case, the abstraction is just the identity function, and the abstract operators are the concrete operators extended to sets of booleans element-wise.

A variable may refer to heap allocated data. Every time an object is allocated, the $record$ function is used to create a heap context (see Section 5). The analysis will keep track of an abstract object for each heap context, which thus play the role of abstract addresses. Formally abstract addresses are $\mathcal{P}$(**HContext**) where **HContext** depends on the chosen analysis variation (see Section 5). We require the existence of a total function $label$: **HContext** $\rightarrow$ **Label** from heap context to allocation site label. Theoretically, this reduces the freedom one has in choosing the $record$ (and $merge$) functions. Practically, this choice does not prevent us from specifying all of the common analysis variations.

An abstract value is then a tuple where each component describes a different type of (abstract) value:

$$v \in \mathbf{Value} = \mathcal{P}(\mathbf{HContext}) \times \mathcal{P}(\mathbf{Bool}) \times \mathcal{P}(\mathbf{Sign}) \times \mathbf{Null}$$

where the abstract values for representing null values **Null** are $\{\top, \bot\}$, where $\top$ represents $null$. For example, $(\bot, \bot, \{+\}, \top)$ represents concrete values that are either a positive integer, or the $null$ value.

We assume the definition of a family of auxiliary functions for injecting individual components, into the lattice, e.g.,

$$inject_{\mathcal{P}(\mathbf{Sign})} : \mathcal{P}(\mathbf{Sign}) \rightarrow \mathbf{Value}$$
$$inject_{\mathcal{P}(\mathbf{Sign})}(l) = (\bot, \bot, l, \bot)$$

Vice versa, PHP silently coerces any value into a value of a suitable type when needed, and our type analysis must mimic this behaviour. Therefore we need a family of $coerce$ functions. The $coerce$ function takes an abstract value and coerces it into a value of a particular type (but taking into account all its abstract components). For example, to coerce an abstract value to a set of signs we employ

$$
\begin{array}{l}
coerce_{\mathcal{P}(\mathbf{Sign})} : \mathbf{Value} \rightarrow \mathcal{P}(\mathbf{Sign}) \\
coerce_{\mathcal{P}(\mathbf{Sign})}(\gamma s, bs, ss, n) = \\
\quad \mathbf{let} \begin{array}[t]{lcl} boolToSign(\texttt{true}) & = & \texttt{+} \\ boolToSign(\texttt{false}) & = & \texttt{0} \\ nullToSign(\top) & = & \{\texttt{0}\} \\ nullToSign(\bot) & = & \emptyset \end{array} \\
\quad \mathbf{in}\ \{\ \texttt{+} \mid \gamma \in \gamma s\ \} \cup \{\ boolToSign(b) \mid b\ \in bs\} \\
\qquad\quad \cup\ ss\ \cup nullTosign(u)
\end{array}
$$

which tells us, for example, that `true` and abstract addresses are interpreted as positive integers, and $null$ and `false` as `0`. We omit all other injection and coercion functions.

Abstract values can be mapped to a *set* of types by the function

$$
\begin{array}{l}
type : \mathbf{Value} \rightarrow \mathcal{P}(\mathbf{Type}) \\
type\ (\gamma s, bs, ss, n) = \\
\quad \mathbf{let} \begin{array}[t]{lcl} type_{\mathbf{Address}}\ (\gamma s) & = & \{\ className_*(label(\gamma)) \mid \\ & & \qquad\qquad \gamma \in \gamma s\ \} \\ type_{\mathbf{Boolean}}\ (bs) & = & \{\ \text{Boolean} \mid b \in bs\ \} \\ type_{\mathbf{Sign}}\ (ss) & = & \{\ \text{Integer} \mid s \in ss\ \} \\ type_{\mathbf{Null}}\ (\top) & = & \{\ \text{Null}\ \} \\ type_{\mathbf{Null}}\ (\bot) & = & \emptyset \end{array} \\
\quad \mathbf{in} \bigcup \{\ type_{\mathbf{Address}}(\gamma s),\ type_{\mathbf{Boolean}}(bs), \\
\qquad\qquad type_{\mathbf{Sign}}(ss),\ type_{\mathbf{Null}}(n)\ \}
\end{array}
$$

where **Type** $=$ **ClassName**$_*$ $\cup$ $\{$ Integer, Boolean, Null $\}$, and the $className_*$ and $label$ functions are used to translate a heap context to a class name. The $label$ function depends on the chosen analysis variation (see later in this paper). Consider the abstract value $v = (\bot, \{\texttt{true}\}, \{+\}, \bot)$ which signifies that the corresponding concrete value is either a positive integer or a Boolean true value. Applying the $type$ function to this value results in a union type: $\{$ Boolean, Integer $\}$.

The analysis operates on abstract states consisting of an abstract stack and an abstract heap, formalized as follows:

$$
\begin{array}{l}
\sigma \in \mathbf{State} = \mathbf{Stack}\ \times\ \mathbf{Heap} \\
H \in \mathbf{Heap} = (\mathbf{HContext}\ \times\ \mathbf{FieldName}_*)\ \mapsto\ \mathbf{Value} \\
H ::= [\,] \mid H[(\gamma, f) \mapsto v] \\
S \in \mathbf{Stack} = \mathbf{Ident}\ \mapsto\ \mathbf{Value} \\
S ::= [\,] \mid S[z\ \mapsto\ v] \\
z \in \mathbf{Ident} = \mathbf{Var}_* \cup \mathbf{Z} \cup \{\mathbb{R}, \mathbb{T}\}
\end{array}
$$

The value $\mathbb{R}$ is a placeholder for the return value of a method invocation, and $\mathbb{T}$ for the receiver object. We use elements of **Z** for representing the values associated with actual parameters in a method call, and which are mapped later to formal parameters. Since the maximum number of parameters in bounded in a given program, the set **Ident** remains finite. Although, formally, S is a list we take the liberty to view it as a finite mapping whenever convenient. We extend **State** to **State**$_\perp$ with a least element $\perp$ to represent unreachable code.

The transfer functions that make up the dynamic part of our analysis operate on abstract states using eleven state manipulation primitives which act as an interface. The first seven ($empty$, $read$, $write$, $writeHeap$, $readHeap$, $readField$ and $writeField$) are used for the intraprocedural transfer functions while $toParameters$, $toVariables$, $clearStack$ and $clearHeap$ are used for the interprocedural transfer functions. We provide only type signatures for the primitives in Figure 4 [21].

The $read$ function reads an abstract value from the abstract stack given an identifier. The $readField$ function returns an abstract value given an identifier and a field name. First an abstract value is read from the stack, given the identifier. This abstract value may refer to multiple heap context elements. For each heap context element an abstract value is read from the heap using the auxiliary function $readHeap$. The return value of the $readField$ function is obtained by joining these abstract values. Following PHP semantics, all read functions return an abstract $null$ if a binding cannot be found. The $write$, $writeField$ and $writeHeap$ are analogous to the versions for reading. In the case of $writeField$ it is important to note that multiple heap context elements on the abstract heap may be updated (because the abstract value read for z may refer to multiple heap context elements). In the case of $writeHeap$, if the heap already contains a value for the given pair (γ, f) the previous and the new value are joined together.

The $toParameters$ function translates the variables to parameter positions. For each variable in $\vec{p}$ a new binding is created between the parameter position and the value of p_i in the abstract stack. The $toVariables$ function translates parameter positions to variables. For each variable in $\vec{p}$ a new binding is created between the variable p_i and the value corresponding to the parameter position i in the abstract stack. Additionally, the special $this$ identifier $\mathbb{T}$ is propagated. The $clearStack$ and the $clearHeap$ functions clear the stack and heap components of the abstract state.

4.2 The analysis itself

We specify the analysis as an instance of the Extended Monotone Framework (as described in detail in [21]). It is a 7-tuple

$$(\textbf{State}_\perp, \mathcal{F}_{State}, init(P_*), \iota^{TA}, f^{TA}_{l,\delta}, \phi^{TA}_{l,\delta}, next^{TA}_{l,\delta}),$$

of which the first five components (the lattice, the function space for transfer functions, the extremal labels, the extremal value, and the transfer functions, respectively), are also part of the Embellished Monotone Frameworks. The components $\phi^{TA}_{l,\delta}$ and $next^{TA}_{l,\delta}$ are specific to our framework, and describe the mutually recursive relationship between intraprocedural and interprocedural flow. They allow us to discover new edges during fix point iteration, and add them to the control flow graph (both intra and inter).

For reasons of space, we focus on what is different with respect to [16, Section 2.5]. As usual, with every program point (l, δ) we associate two values $A_\circ(l, \delta)$ and $A_\bullet(l, \delta)$. The former collects all the information flowing in from statements that precede it (in the analysis flow direction, forward in this case), and the latter describes what holds after the elementary block l has been executed. The formulas for $A_\circ(l, \delta)$ are those of [16].

The *extremal value* ι^{TA} denotes that the extremal program points are reachable: $\iota^{TA} = empty$ (recall that $empty$ is one of our primitives).

The major differences with our Extended Monotone Frameworks is that as part of the analysis we also iteratively compute the intraprocedural F and interprocedural IF flow edges. The formulas are as follows:

$$\begin{aligned}
&F \in \textbf{Flow} = \mathcal{P}(\textbf{Point} \times \textbf{Point})\\
&F = \{\, next^{TA}_{e,\Lambda}(\emptyset) \mid e \in E\} \cup\\
&\qquad \{\, next^{TA}_{l',\delta'}(IF) \mid ((l,\delta),(l',\delta')) \in F\,\}
\end{aligned}$$

$$\begin{aligned}
&IF \in \textbf{InterFlow} = \mathcal{P}(\textbf{Point} \times \textbf{Point} \times \textbf{Point} \times \textbf{Point})\\
&IF = \phi^{TA}_{l,\delta}(A_\bullet(l,\delta)) \cup IF \text{ for all}(l,\delta) \text{ in } F
\end{aligned}$$

In the remainder of the section we shall provide the remaining pieces of the puzzle: the extremal value ι^{TA}, the transfer functions $f^{TA}_{l,\delta}$ and $f^{TA}_{l_c,l_r}$, the dynamic intraprocedural flow function $next^{TA}_{l,\delta}$ and the dynamic interprocedural flow function $\phi_{l,\delta}$.

A transfer function $f^{TA}_{l,\delta} : \textbf{State}_\perp \to \textbf{State}_\perp$ specifies how flow and context sensitive type information flows from the entry to the exit of an elementary block. Data flow information should only be propagated if the elementary block is reachable, so we define

$$f^{TA}_{l,\delta}(\sigma) = \begin{cases} \perp & \text{if } \sigma = \perp \\ \psi_{l,\delta}(\sigma) & \text{otherwise} \end{cases}$$

that delegates the actual work to the function $\psi_{l,\delta} : \textbf{State} \to \textbf{State}$. Procedure return is exceptional in that we need to combine two flows. Therefore we need a binary transfer function $f^{TA}_{l_c,l_r} : \textbf{State}_\perp \to \textbf{State}_\perp \to \textbf{State}_\perp$. The first parameter describes the data flow information before the method call and the second parameter describes the data flow information at the exit of the method body. It is defined similarly to the unary version:

$$f^{TA}_{l_c,l_r}(\sigma,\sigma') = \begin{cases} \perp & \text{if } \sigma = \perp \text{ or } \sigma' = \perp \\ \psi_{l_c,l_r}(\sigma,\sigma') & \text{otherwise} \end{cases}$$

For each kind of elementary block we define the necessary ψ functions in Figures 5 and 6. Most of these should be self evident, but some remarks may be useful in understanding: In the rules for booleans and numbers, we have to apply the correct function to inject the value into the state. In the rule for binary operators, we apply the lifted operator $\widetilde{\odot}$ that "simulates" $\odot$ in the abstract domain. Neither $skip$, boolean tests and method exits affect the state, so we use the identity function there. In the case of a new statement we invoke the $record$ function that creates a new heap context. We will describe several variations in Section 5.

The transfer function for a method call (the first case in Figure 6) implements a part of the parameter passing semantics. The transfer function for a method call translates the identifiers corresponding to actual parameters to parameter position by means of the $toParameters$ function. Subsequently, the transfer function for a method entry translates the parameter positions to the formal parameter identifiers using the $toVariables$ function. The special $this$ identifier, $\mathbb{T}$, is used to make the receiver object available in the callee. The transfer function for a method entry implements the second half of the parameter passing semantics by translating the parameter positions to the formal parameters. The transfer function for $return$ propagates the abstract value stored in the identifier v to the special return identifier $\mathbb{R}$. The only binary transfer function ψ_{l_c,l_r} for return, reads the return value, stores it, and ensures the new state combines the stack component of the caller with the heap components of the callee.

Upon each method call new edges may be added to the interprocedural flow, IF. Each edge in the interprocedural flow represents a method invocation which may occur in the program under analysis. Each interprocedural flow edge is specified by a tuple consisting of four program points: the call position of the caller, the entry of the callee, the exit of the callee and the return position of the caller. For a given method invocation $[v = v'.m\ (\vec{p})]^{l_c}$, the $\phi^{TA}_{l_c,\delta}$ function

$$\begin{array}{l}
empty : \textbf{State} \\
read : \textbf{Ident} \times \textbf{State} \rightarrow \textbf{Value} \\
write : \textbf{Ident} \times \textbf{Value} \times \textbf{State} \rightarrow \textbf{State} \\
readField : \textbf{Ident} \times \textbf{FieldName} \times \textbf{State} \rightarrow \textbf{Value} \\
readHeap : \textbf{HContext} \times \textbf{FieldName} \times \textbf{Heap} \rightarrow \textbf{Value} \\
writeField : \textbf{Ident} \times \textbf{FieldName} \times \textbf{Value} \times \textbf{State} \rightarrow \textbf{State} \\
writeHeap : \textbf{HContext} \times \textbf{FieldName} \times \textbf{Value} \times \textbf{Heap} \rightarrow \textbf{Heap} \\
toParameters : \mathcal{P}(\textbf{Var}_*) \times \textbf{State} \rightarrow \textbf{State} \\
toVariables : \mathcal{P}(\textbf{Var}_*) \times \textbf{State} \rightarrow \textbf{State} \\
clearStack : \textbf{State} \rightarrow \textbf{State} \\
clearHeap : \textbf{State} \rightarrow \textbf{State}
\end{array}$$

Figure 4. The signatures of the eleven manipulation primitives

$$\begin{array}{lcll}
[skip]^l & : & \psi_{l,\delta}(\sigma) = & \sigma \\
[b]^l & : & \psi_{l,\delta}(\sigma) = & \sigma \\
[v = b]^l & : & \psi_{l,\delta}(\sigma) = & write(v, inject_{\mathcal{P}(\textbf{Bool})}(\{b\}), \sigma) \\
[v = n]^l & : & \psi_{l,\delta}(\sigma) = & write(v, inject_{\mathcal{P}(\textbf{Sign})}(fromInteger(n)), \sigma) \\
[v = v']^l & : & \psi_{l,\delta}(\sigma) = & write(v, read(v', \sigma), \sigma) \\
[v = v' \odot v'']^l & : & \psi_{l,\delta}(\sigma) = & write(v, read(v', \sigma) \mathbin{\widetilde{\odot}} read(v'', \sigma), \sigma) \\
[v = \texttt{new}\ C]^l & : & \psi_{l,\delta}(\sigma) = & \textbf{let}\ \gamma \quad = \quad record(l, \delta) \\
 & & & \quad\ value \quad = \quad inject_{\mathcal{P}(\textbf{HContext})}(\{\gamma\}) \\
 & & & \textbf{in}\ write(v, value, \sigma) \\
[v.f = v']^l & : & \psi_{l,\delta}(\sigma) = & writeField(v, f, read(v', \sigma), \sigma) \\
[v = v'.f]^l & : & \psi_{l,\delta}(\sigma) = & write(v, readField(v', f, \sigma), \sigma)
\end{array}$$

Figure 5. Intraprocedural transfer functions

$$\begin{array}{lcll}
[v = v'.method(\vec{p})]^{l_c} & : & \psi_{l_c,\delta}(\sigma) = & \textbf{let}\ value \ = \ read(v', \sigma) \\
 & & & \quad\ \sigma' \ = \ toParameters(\vec{p}, \sigma) \\
 & & & \textbf{in}\ write(\mathbb{T}, value, \sigma') \\
[C.method(\vec{p})]^{l_n} & : & \psi_{l_n,\delta}(\sigma) = & toVariables(\vec{p}, \sigma) \\
[\texttt{return}\ v]^l & : & \psi_{l,\delta}(\sigma) = & \textbf{let}\ value \ = \ read(v, \sigma) \\
 & & & \textbf{in}\ write(\mathbb{R}, value, \sigma) \\
[C.method(\vec{p})]^{l_x} & : & \psi_{l_x,\delta}(\sigma) = & \sigma \\
[v = v'.method(\vec{p})]^{l_r} & : & \psi_{l_c,l_r}(\sigma, \sigma') = & \textbf{let}\ value \ = \ read(\mathbb{R}, \sigma') \\
 & & & \quad\ \sigma'' \ = \ clearHeap(\sigma) \sqcup clearStack(\sigma') \\
 & & & \textbf{in}\ write(v, value, \sigma'')
\end{array}$$

Figure 6. The interprocedural transfer functions

retrieves the set of heap context elements to which the identifier v' may point. An edge will be added to the interprocedural flow for each heap context element. Dynamic dispatch is resolved at runtime, so depending on the heap context element different method definitions may be called. The $resolve_*$ function is used to traverse the inheritance hierarchy and locate the targeted method definition. The $merge$ function (see Section 5) combines the call label l_c, a heap context element γ and the current analysis context δ and returns the context under which the callee will be analyzed (for all other elementary blocks $\phi^{TA}_{l\delta}(\sigma) = \emptyset$), see Figure 7).

Adding new intraprocedural edges with $next^{TA}_{l,\delta}$

Whenever a new interprocedural edge to some method is added, it may well be necessary to add additional interprocedural edges. This is the role of the $next^{TA}_{l,\delta}$ function. We need to distinguish three cases.

For a method invocation $[v = v'.method(\vec{p})]^{l_c}$, information needs to propagate from the caller to the entry of the callee. The interprocedural flow, IF, tells us to which callee and to which context the data flow information needs to propagate.

$$\begin{array}{l}
next_{l_c,\delta}(IF) = \\
\quad \{((l_c, \delta), (l_n, \delta')) \mid ((l_c, \delta), (l_n, \delta'), (l_x, \delta'), (l_r, \delta)) \in IF\ \} \\
\qquad \cup \\
\quad \{((l_x, \delta'), (l_r, \delta)) \mid ((l_c, \delta), (l_n, \delta'), (l_x, \delta'), (l_r, \delta)) \in IF\ \}
\end{array}$$

In the case of $[end^{l_x}]$, information needs to propagate back to the caller at the end of a method body. To avoid poisoning information should only flow back to the original context under which the caller was being analyzed:

$$\begin{array}{l}
next_{l_x,\delta}(IF) = \\
\quad \{((l_x, \delta), (l_r, \delta')) \mid ((l_c, \delta'), (l_n, \delta), (l_x, \delta), (l_r, \delta')) \in IF\ \}
\end{array}$$

Finally, if l corresponds to any other elementary block the information will be propagated following the standard intraprocedural

$$
\begin{array}{llll}
\phi^{TA}_{l_c,\delta}(\sigma) = \textbf{let } \gamma s & = & \multicolumn{2}{l}{coerce_{\mathcal{P}(\textbf{HContext})}(read(v', \sigma))} \\
\quad edge(\gamma) & = & \textbf{let } \delta' = & merge(l_c, \gamma, \delta) \\
& & \quad\tau = & className_*(label(\gamma)) \\
& & \quad m_r = & resolve_*((\tau, m)) \\
& & \quad l_r = & return_*(l_c) \\
& & \multicolumn{2}{l}{\textbf{in}((l_c, \delta), (entry_*(m_r), \delta'), (exit_*(m_r), \delta'), (l_r, \delta))} \\
\textbf{in } \{\ edge(\gamma) \mid \gamma \in \gamma s\ \} & & &
\end{array}
$$

Figure 7. The function $\phi^{TA}_{l_c,\delta}$

flow, under the same context:

$$next_{l,\delta}(IF) = \{((l, \delta), (l', \delta)) \mid (l, l') \in flow_*\ \}$$

On soundness

Since there is no formal specification of PHP, the soundness of our implementation can only be established by comparing the inferred types of our type analysis to the observed types while running the program. To cover all execution paths a set of unit tests was written. The inferred type set is obtained by running the type analysis and transforming the calculated abstract values to type sets (using $type$, see Section 4).

The run-time type sets are obtained by instrumenting the original source code of the programs listed in Section 6. On each assignment the run-time type of the assigned variable is obtained by means of the $gettype$ function. If the resulting type constitutes an object or a resource the type is further refined by calling get_class and $get\_resource\_type$ respectively. This results in a run-time type set since each assignment may be executed multiple times with possibly different values. Hence different types may be observed.

We then compared the observed run-time types with the inferred type sets by our implementation. We found that for all assignments in each program the types we observed formed a subset of the inferred type sets. In other words, the type analysis was able to infer all types observed at run-time, providing an empirical approximation to soundness. Note, though that it is as good as the unit tests we have written. As we shall see later (see Figure 12), for many of the assignments we *inferred* the exact same set of types as we *observed dynamically*.

5. Object-sensitive type analysis

Object-sensitivity is particularly well suited context abstraction for analysing object-oriented programs. However, an object-sensitive analysis has many degrees of freedom relating to which context elements are selected upon each method invocation or object allocation. In this paper we use the framework of [20] to describe various forms of object-sensitivity, borrowing their terminology of full-object-sensitivity, plain-object-sensitivity and type-sensitivity. Their framework offers a clean model to design and reason about different analysis variations. An analysis variation is given by defining the two context manipulation functions:

$$record : \textbf{Label} \times \textbf{Context} \to \textbf{HContext}$$
$$merge : \textbf{Label} \times \textbf{HContext} \times \textbf{Context} \to \textbf{Context}$$

Every time an object is allocated, the $record$ function is used to create a heap context. The heap context is stored and used as an abstraction of the allocated object. The $merge$ function is used on every method invocation. The call site label, the heap context of the receiver object and the current context are merged to obtain the context in which the invoked method will be executed. The type analysis described in Section 4 uses these two context manipulation functions in some of its transfer functions ($record$ is used for the new block, and the $merge$ and $label$ functions are employed by $\phi^{TA}_{l_c,\delta}$). The $label$ function is not a context manipulation function but rather an artefact of our type analysis which requires the existence of a total function from heap context to allocation site label.

A *full-object-sensitive* analysis will analyze every dispatched method under the heap context associated with the receiver object. Since the heap context consists of multiple allocation site labels, the hope is that these labels split the data flow facts into separate sets of facts that are widely different. Milanova et al. [15] was the first to use allocation site labels for this purpose. We can specify a concrete full-object-sensitive analysis by defining the $record$, $merge$ and $label$ functions as follow:

$$
\begin{aligned}
\textbf{Context} &= \textbf{Label}^n \\
\textbf{HContext} &= \textbf{Label}^m \\
record(l, \delta) &= first_m(cons(l, \delta)) \\
merge(l, \gamma, \delta) &= first_n(\gamma) \\
label(\gamma) &= car(\gamma)
\end{aligned}
$$

The context under which a method will be analyzed depends on the heap context of the receiver object. The heap context of the receiver object depends on (1) its allocation site label and (2) the context under which the receiver object was allocated. So, the context under which a method is analyzed depends on the allocation site label of the receiver, the allocation site label of the object that allocated the receiver, the allocation site label of the object that allocated the object that allocated the receiver, and so on.

We follow [20] in naming the common analysis variations: for a full-object-sensitive analysis with a regular context depth of n and a heap context depth of $m+1$ we write $n$full+mH. In order to make the set of heap contexts finite we limit the number of allocation site labels in a context element to a fixed number (in practice 2). Hence, 1full, 1full+1H and 2full+1H are of particular interest.

Plain-object-sensitivity

In contrast to full-object-sensitivity, a plain-object-sensitive analysis combines both the heap context of the receiver object and the regular context of the caller:

$$
\begin{aligned}
\textbf{Context} &= \textbf{Label}^n \\
\textbf{HContext} &= \textbf{Label}^m \\
record(l, \delta) &= first_m(cons(l, \delta)) \\
merge(l, \gamma, \delta) &= first_n(cons(car(\gamma), \delta)) \\
label(\gamma) &= car(\gamma)
\end{aligned}
$$

Both full-object-sensitive and plain-object-sensitive analyses store allocation site labels as context elements using the $record$ function. The distinction lies in the $merge$ function. The $merge$ function decides which elements to keep when a method is invoked: only keep the heap context elements of the receiver object (as in full-object-sensitivity) or merge the heap context of the receiver object with the regular context of the caller (as in plain-object-sensitivity). Paddle [13], for example, uses plain-object-sensitivity.

Smaragdakis et al. [20] found that full-object-sensitivity outperforms plain-object-sensitivity. The explanation they provide is that with plain-object-sensitivity it is more likely that when merging the allocation site label of the receiver object with the allocation site label of the caller object the two labels are more likely to be correlated. For example the receiver and the caller object are exactly the same if an object calls a method on itself. And if they are correlated they will do a worse job at separating the data flow facts.

For a plain-object sensitive analysis with a regular context depth of n and a heap context depth of $m+1$ we shall write $n$plain+mH. Note that 1plain and 1plain+1H coincides with respectively 1full and 1full+1H, so we shall simply denote these analysis variations with 1obj and 1obj+1H.

Type Sensitivity

Generally speaking, the precision of an analysis is improved by separating data flow information depending on the calling context. To achieve termination, we abstract the possibly infinite set of calling contexts to a finite set of abstract contexts δ. But finite is not necessarily small: any given program may have many allocation sites. To overcome the combinatorial explosion of abstract contexts, [20] introduced a type-sensitive analysis is to improve scalability by using a coarser approximation of objects: instead of allocation site labels we approximate an object by its type. Hence a type sensitive analysis is similar to an object sensitive analysis: whereas an object sensitive analysis uses allocation site labels as context elements, a type sensitive analysis uses types as context elements. In the remainder of the section we shall describe two variations on this theme.

A 2-type-sensitive analysis employs a regular context which consists of two types. This reduces the number of possible context elements as the number of types in a program is typically much smaller than the number of allocation sites. For a 2type+1H analysis we define the following context manipulation functions:

$$\textbf{Context} = \textbf{ClassName}^2$$
$$\textbf{HContext} = \textbf{Label} \times \textbf{ClassName}$$
$$record(l, \delta = [C_1, C_2]) = [l, C_1]$$
$$merge(l, \gamma = [l', C], \delta) = [\mathcal{T}(l'), C]$$
$$label(\gamma = [l, C]) = l$$

One may notice that the merge function only uses the heap context of the receiver object and ignores the context of the caller object. In this sense the 2type $+ 1H$ analysis is a variation of the 2full $+ 1H$ analysis, and not of the 2plain $+ 1H$ analysis.

Consider a statement $obj = [newA]^l$ inside a class C. Following [20], the function $\mathcal{T}$: **Label** $\rightarrow$ **ClassName** returns the upper bound C on the dynamic type of the allocator object.

Another choice of context is to replace only one allocation site label with a type. This leads to a 1type1obj $+ 1H$ analysis:

$$\textbf{Context} = \textbf{Label} \times \textbf{ClassName}$$
$$\textbf{HContext} = \textbf{Label}^2$$
$$record(l, \delta = [l', C_2]) = [l, l']$$
$$merge(l, \gamma = [l_1, l_2], \delta) = [l_1, \mathcal{T}(l_2)]$$
$$label(\gamma = [l_1, l_2]) = l_1$$

This choice of context is interesting, because we expect the number of context elements to be in between that of a 2full $+ 1H$ analysis and a 2type $+ 1H$ analysis.

6. Evaluation

We have implemented and evaluated several of the analysis variations, as described in Section 5, up to a context depth of 2. Our implementation consists of two distinct phases. In the first phase, an intermediate representation is obtained by parsing the original PHP program using PHC [1]. In a pipeline of sequential transformations PHC lowers the original AST to various intermediate forms, in our case to an intermediate representation called Higher Internal Representation (HIR). This is the last phase in which the result of the transformation is still a valid PHP program. In the second phase the HIR is read by the type inferencer which is written in Haskell and the UU Attribute Grammar system.

Project	**Description**	**LoC**
Ray Tracer	A PHP implementation of a ray tracer. Ray tracing is a technique to generate an image of a 3D scene by tracing a ray of light through the image plane and simulating the effects of each object it intersects.	915
Gaufrette	Gaufrette is a file system abstraction layer, which allows an application developer to develop an application without knowing where the files are stored and how. Gaufrette offers support for various file systems like Amazon S3 and Dropbox.	2974
PHPGeo	PHPGeo provides an abstraction to different geographical coordinate systems and allows an application developer to calculate distances between different coordinates.	1634
MIME	A MIME library which allows an application developer to compose and send email messages according to the MIME standard [2].	486
MVC	A framework which implements the model-view-controller pattern for web application.	2583
Dijkstra	An implementation of Dijkstra's algorithm [5] using adjacency lists to represent a graph structure.	4854
Floyd	An implementation of the Floyd-Warshall algorithm [7] using an adjacency matrix to represent a graph structure.	5742
Interpreter	An object-oriented implementation of a small expression language, including a parser.	843

Figure 8. List of projects in the test suite

The PHP programs in the test suite are shown in Figure 8. It was necessary to make small modifications to the original programs on some occasions due to the use of unsupported language features (see Section 2). These modifications are documented in a file called `modifications.txt`, which is present in the directory of each project.

The source code (in Haskell) and the test suite are publicly available from
`http://www.github.com/henkerik/objectsensitivetyping/`.
The experiments were performed on a machine with a Intel Core 2 Duo 3.0Ghz processor with 3.2GiB of internal memory running Ubuntu 12.04.

Results

We shall compare the precision of plain-object sensitivity to full-object sensitivity. Theoretically, we expect that a full object sensitive analysis shall give a better precision for the same context depth. We included the results of a context-insensitive, a 1obj sensitive and a 1obj+1H sensitive analysis for comparison. For each analysis variation we collected the following set of precision and performance metrics:

- **# of union types** shows the number of assignments for which the type analysis could not infer a single type. Note that due to the dynamic nature of PHP it is not always possible to infer a single type.
- **# of union types collapsed** shows the number of assignments for which the type analysis could infer a single type after collapsing object types with a common ancestor. Moreover, the $Null$ type is ignored if the remaining type set only contains class names.
- **# of polymorphic call sites** shows the number of method call sites for which the type analysis could not infer a unique receiver method.
- **# of call graph edges** shows the number of call graph edges.
- **average var points-to** shows the average number of allocation sites to which a variable can refer.
- **execution time** shows the average running time for 20 executions of the implementation. We used Criterion [1] to obtain the execution time.

We shall illustrate the concept of collapsing types with a common ancestor. Consider a program with two classes named `Add` and `Minus` with a common parent class `Expr`. The following table shows various type sets and their collapsed counter parts:

Un-collapsed Types	**Collapsed Types**
{ Boolean, Integer }	{ Boolean, Integer }
{ Boolean, Null }	{ Boolean, Null }
{ Add, Minus }	{ Expr }
{ Add, Null }	{ Add }

Table 1. Collapsing types

In Figure 9 and 10 we provide the main results of our experiments. All metrics in these tables are end-user (i.e. context-insensitive) metrics. This means that the analysis result for different contexts are joined together for the same program label.

We shortly summarise our findings. In terms of precision, 2plain $+ 1H$ and 2full $+ 1H$ do not differ at all, except for `raytracer` where 2full $+ 1H$ analysis achieves better precision. We expect these small differences to be due to the relatively small size of the programs. In terms of performance, 2full $+ 1H$ always either outperforms the 2plain $+ 1H$ analysis or both analyses end up taking a similar amount of time. Interestingly, increasing the context depth does not necessarily result in a performance penalty. For example, the context insensitive analysis (which only uses one context Λ) performs significantly worse than the more complicated 2full $+ 1H$ analysis for 6 of the 8 test programs. This difference is most striking in the case of the *phpgeo* test program where the context insensitive analysis is more than 7 $\times$ slower than the 2full$+1H$ analysis. We suspect the main reason is that the higher precision enables the analysis to exclude a broader range of target methods while resolving a method call.

[1] http://hackage.haskell.org/package/criterion

In terms of performance, 2type $+ 1H$ only outperforms the 2full $+ 1H$ analysis for 3 of 8 test programs. Compared to the 2full $+ 1H$ analysis, the 1type1obj+1H analysis does not perform better for a single test program. Contrary to our expectations the type sensitive analysis often performs worse than a full object sensitive analysis of the same context depth.

If we increase the context depth of an analysis, the execution time is influenced by two opposing forces. On the one hand a deeper context depth may result in each data flow fact being analyzed more often, leading to an increase in the execution time. On the other hand, a deeper context may avoid poisoning of the analysis results. This prevents the propagation of data flow facts because the analysis is able to infer statically that some program paths are impossible, leading to a decrease in the execution time.

The relative strength of these two forces depends strongly on the specific implementation decisions. We suspect that our implementation differs in this regard with that used in [20], leading to different experimental observations. Consider for example the extreme case of an context insensitive analysis, which employs only one context Λ. The context insensitive analysis performs worse in terms of performance than the 2full $+ 1H$ analysis for 6 out of the 8 test programs in our experiments. However, the context insensitive analysis performs better in terms of performance than the 2full $+ 1H$ analysis for all test programs in the experiments described in [20].

Since the number of contexts of a type sensitive analysis lies in between the number of contexts of a context insensitive analysis (only one context) and a 2full $+ 1H$ analysis (theoretically $\mathcal{O}(n^2)$ number of contexts, where n is the number of allocation sites in a program) one may expect a performance increase using the implementation of Smaragdakis et al. while a performance decrease is expected using our implementation.

In terms of precision, 2type $+ 1H$ performs worse than 2full+1 for 3 out of 8 test programs. Only for the $raytracer$ test program the 1type1obj+1H analysis performs worse in terms of precision, for the others results are identical.

Abstract Garbage Collection

Our experiments show that the type analysis only terminates within a reasonable amount of time if abstract garbage collection [14] is enabled. Abstract garbage collection prevents the propagation of abstract objects which are known to be unreachable. Table 11 shows the execution time of the analysis with and without abstract garbage collection. We ran this experiment only on a subset of the test suite. Programs excluded for this experiment ran out of memory when abstract garbage collection was disabled.

	GC Enabled (s)	GC Disabled (s)
mime	0.40	0.59
raytracer	6.48	13.56
interpreter	2.33	5.36

Figure 11. Performance Metrics of Abstract Garbage Collection

Exact inferred type sets

Consider the table in Figure 12. It provides for each program, and for each analysis variation, the number of assignments reachable in the program (RA), and the number of assignments among those for which our the analysis variant inferred exactly the same set of types as those observed in our unit tests (PM = precise match), and the relative portion of the assignments for which this is the case. For example, for $raytracer$, the insensitive variant discovered 730 reachable assignments, and inferred for 360 (i.e, 49 percent) type sets that exactly matched the dynamically observed types for that

		insensitive	1obj	1obj+1H	2plain+1H	2full+1H
raytracer	# of union types	324	-84	0	-6	-12
	# of union types coll.	213	-28	0	-6	-12
	# of poly. call sites	26	-22	0	0	0
	# of callgraph edges	155	-28	0	0	0
	average var points-to	11.24	1.47	1.47	1.47	1.47
	execution time (s)	8.81	5.43	7.43	7.00	6.02
gaufrette	# of union types	141	-12	4	0	0
	# of union types coll.	69	-7	4	0	0
	# of poly. call sites	8	-1	0	0	0
	# of callgraph edges	234	-1	0	0	0
	average var points-to	3.43	2.36	2.36	2.36	2.36
	execution time (s)	4.22	2.97	3.44	3.45	3.09
phpgeo	# of union types	164	-22	0	0	0
	# of union types coll.	119	-25	0	0	0
	# of poly. call sites	52	-52	0	0	0
	# of callgraph edges	244	-108	0	0	0
	average var points-to	14.60	1.69	1.69	1.69	1.69
	execution time (s)	14.74	3.65	4.74	1.94	1.94
mime	# of union types	62	0	-5	0	0
	# of union types coll.	28	0	-5	0	0
	# of poly. call sites	2	0	0	0	0
	# of callgraph edges	49	0	0	0	0
	average var points-to	2.47	1.12	1.07	1.07	1.07
	execution time (s)	0.45	0.43	0.43	0.51	0.51
mvc	# of union types	179	-47	1	0	0
	# of union types coll.	110	-57	0	0	0
	# of poly. call sites	36	-27	0	0	0
	# of callgraph edges	301	-143	0	0	0
	average var points-to	8.16	1.44	1.09	1.09	1.09
	execution time (s)	12.59	4.60	5.81	5.70	5.20
dijkstra	# of union types	128	-1	-8	0	0
	# of union types coll.	61	-2	-36	0	0
	# of poly. call sites	3	0	-2	0	0
	# of callgraph edges	144	0	-12	0	0
	average var points-to	3.74	2.05	1.31	1.31	1.31
	execution time (s)	12.15	6.75	4.47	3.84	3.36
floyd	# of union types	150	1	-4	0	0
	# of union types coll.	42	0	-15	0	0
	# of poly. call sites	9	0	-2	0	0
	# of callgraph edges	176	0	-9	0	0
	average var points-to	5.15	1.75	1.51	1.50	1.50
	execution time (s)	18.87	13.17	13.73	12.90	11.80
interpreter	# of union types	241	-12	0	0	0
	# of union types coll.	92	-2	0	0	0
	# of poly. call sites	59	0	0	0	0
	# of callgraph edges	495	0	0	0	0
	average var points-to	5.05	3.90	3.90	3.90	3.90
	execution time (s)	2.03	2.05	2.10	2.95	2.09

Figure 9. Comparison of plain and full object sensitivity

		1obj+1H	2type+1H	1type1obj+1H	2full+1H
raytracer	# of union types	240	56	-56	-18
	# of union types coll.	185	6	-6	-18
	# of poly. call sites	4	14	-14	0
	# of callgraph edges	127	14	-14	0
	average var points-to	1.47	3.86	1.47	1.47
	execution time (s)	7.43	7.29	7.65	6.02
gaufrette	# of union types	133	4	-4	0
	# of union types coll.	66	0	0	0
	# of poly. call sites	7	0	0	0
	# of callgraph edges	233	0	0	0
	average var points-to	2.36	2.84	2.36	2.36
	execution time (s)	3.44	4.00	3.39	3.09
phpgeo	# of union types	142	4	-4	0
	# of union types coll.	94	0	0	0
	# of poly. call sites	0	40	-40	0
	# of callgraph edges	136	96	-96	0
	average var points-to	1.69	4.26	1.69	1.69
	execution time (s)	4.74	4.56	2.22	1.94
mime	# of union types	57	0	0	0
	# of union types coll.	23	0	0	0
	# of poly. call sites	2	0	0	0
	# of callgraph edges	49	0	0	0
	average var points-to	1.07	1.88	1.07	1.07
	execution time (s)	0.43	0.45	0.52	0.51
mvc	# of union types	133	27	-27	0
	# of union types coll.	53	25	-25	0
	# of poly. call sites	9	13	-13	0
	# of callgraph edges	158	26	-26	0
	average var points-to	1.09	2.54	1.09	1.09
	execution time (s)	5.81	4.06	5.49	5.20
dijkstra	# of union types	119	0	0	0
	# of union types coll.	23	0	0	0
	# of poly. call sites	1	0	0	0
	# of callgraph edges	132	0	0	0
	average var points-to	1.31	1.77	1.31	1.31
	execution time (s)	4.47	4.82	4.45	3.36
floyd	# of union types	147	0	0	0
	# of union types coll.	27	0	0	0
	# of poly. call sites	7	0	0	0
	# of callgraph edges	167	0	0	0
	average var points-to	1.51	4.31	1.51	1.50
	execution time (s)	13.73	16.44	13.57	11.80
interpreter	# of union types	229	0	0	0
	# of union types coll.	90	0	0	0
	# of poly. call sites	59	0	0	0
	# of callgraph edges	495	0	0	0
	average var points-to	3.90	4.70	3.90	3.90
	execution time (s)	2.10	1.90	2.10	2.09

Figure 10. Comparison of type sensitivity and object sensitivity

assignment (by running the program on a number of unit tests, as described at the end of Section 4). For the 2full $+$ $1H$ case, one fewer assignment was found to be reachable, and the percentage improved to 63 percent.

If we assume our analysis to be sound, the assignments that have a precise match cannot be improved upon. Although the variation between the ratio's for a given program are never very high, and for some programs they are almost identical across different variations, it does show that the more precise variants can sometimes give substantially better numbers, and that on the whole the percentages are not bad at all.

7. Related Work

The concept of soft typing was introduced in [4]. A soft type system accepts all programs in a dynamically typed language and inserts dynamic checks in places where it cannot statically infer provably correct types. Flanagan introduces hybrid type checking, which is a synthesis of static typing and dynamic contract checking [6]. Gradual typing, see, e.g., [19] allows mixing static and dynamic typing within one program: type annotated program elements are checked statically, others dynamically.

There is quite a bit of work on soft typing for particular languages, including PHP, Python, and JavaScript. Due to reasons of space we can only provide details for a few of these. The works discussed here provide additional pointers to related work. Jensen et al. [10] presents a static program analysis to infer detailed and sound type information for Javascript programs by means of abstract interpretation. Their analysis is both flow and context sensitive and supports the full language, as defined in the ECMAScript standard, including its prototypical object model, exceptions and first-class functions. The analysis results are used to detect programming errors and to produce type information for program comprehension. The precision of the analysis is improved by employing a technique called recency abstraction. This enables the analysis to perform strong updates on this object, keeping the abstraction as precise as possible.

Fritz et al. [8] performs type analysis for Python programs, focusing on balancing precision and cost by controlling a widening operator that is employed during fix point iteration. The proposed analysis is based on data flow and is both flow and context sensitive. The analysis supports first class functions and Python's dynamic class system. Both [8] and [10] employ an extension to embellished monotone frameworks that is similar to ours.

Camphuijsen et al. [3] presents a type analysis for PHP as part of a tool to detect suspicious code. The analysis is flow and context sensitive and the type system is based on union types, but also support user-provided polymorphic types for functions. First-class functions and object-oriented programming constructs are not supported by the analysis. A widening operator is used to force termination in the presence of infinitely nested array structures.

Context-sensitivity has a pretty long history, see, e.g., [18] and [17]. Object-sensitivity as a particular form of context-sensitivity was introduced by Milanova et al. [15]. Smaragdakis et al. [20] describe a framework in which it is possible to describe different variations of object sensitivity. Their abstract semantics is parametrized by two functions which manipulate contexts, $record$ and $merge$ that we also employ in our work. They then specify many variations of object-sensitivity by choosing different $record$ and $merge$ functions. They also introduced the concept of type sensitivity as an approximation of object sensitivity. Their work shows that type sensitivity preserves much of the precision of object sensitivity at considerably lower cost.

8. Conclusion and Future Work

In this paper we described an object sensitive type analysis for PHP. The presence of dynamic method dispatching in PHP implies that control flow and data flow information are mutually dependent: propagation of points-to information may make additional methods reachable, which may in turn increase the propagated points-to information.

We specified the type analysis as an extension of a points-to analysis. In addition, we presented a novel method to capture the coercion rules of PHP by means of the $coerce$ and $reject$ functions. We considered multiple variations of an object sensitive analysis, and gave the results of an experimental evaluation of our implementation on eight PHP applications. We did not succeed in achieving gains similar to those in [20]. We did find that abstract garbage collection is essential to decrease memory consumption to an acceptable level.

We do not yet support first class functions and closures that require an even more elaborate lattice to model the abstract state. Following [8], it seems we should be able to base such an extension on monotone frameworks we used in this work. Extensions to improve the precision of our analysis is to employ recency abstraction [10], or path-sensitivity [9].

In [11], the authors observes that most client analyses do not care about the specific exception objects, rather they care about the impact of exceptions on the control flow of a program. This leads to the question whether our type analysis can also benefit from a coarsening of exception objects. Because PHP is also used as a procedural (not object-oriented) language, it also makes sense to apply the hybrid approach of [12] that combines call-site sensitivity with object sensitivity. Finally, we wonder whether the Datalog query language used in the DOOP framework employed by Smaragdakis is expressive enough to describe the coercion rules of PHP.

We shall also report on our extension to embellished monotone frameworks for dealing with the dynamic aspects of PHP, and comparing it with other work in this area in another paper.

References

[1] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1916–1923, New York, NY, USA, 2009. ACM.

[2] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies. RFC 1341 (Proposed Standard), June 1992. Obsoleted by RFC 1521.

[3] P. Camphuijsen, J. Hage, and S. Holdermans. Soft typing php. Technical Report UU-CS-2009-004, Department of Information and Computing Sciences, Utrecht University, 2009.

[4] R. Cartwright and M. Fagan. Soft typing. *PLDI '91: 1991 Conference on Programming Language Design and Implementation*, pages 278–292, Jun 1991.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[6] C. Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM symposium on Principles of programming languages*, pages 245–256, New York, USA, 2006. ACM.

[7] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[8] L. Fritz and J. Hage. Cost versus precision for approximate typing for python. Technical Report UU-CS-2014-017, Department of Information and Computing Sciences, Utrecht University, 2014.

[9] H. Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 52–58. ACM, 2005.

	insensitive			1obj			1obj+1H			2plain + $1H$			2full + $1H$		
	RA	PM	%	RA	PM	%	RA	PM	%	RA	PM	%	RA	PM	%
raytracer	730	360	49 %	729	441	60 %	729	441	60 %	729	447	61 %	729	459	63 %
gaufrette	672	432	64 %	671	436	65 %	671	432	64 %	671	432	64 %	671	432	64 %
phpgeo	699	518	74 %	699	538	77 %	699	538	77 %	699	538	77 %	699	538	77 %
mime	320	211	66 %	320	211	66 %	320	213	67 %	320	213	67 %	320	213	67 %
mvc	566	173	31 %	561	213	38 %	561	221	39 %	561	221	39 %	561	221	39 %
dijkstra	575	321	56 %	575	323	56 %	522	323	62 %	522	323	62 %	522	323	62 %
floyd	798	556	70 %	798	557	70 %	761	559	73 %	761	559	73 %	761	559	73 %
interpreter	555	265	48 %	555	267	48 %	555	267	48 %	555	267	48 %	555	267	48 %

Figure 12. How often does what we statically infer correspond exactly to what is dynamically observed?

[10] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[11] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in java points-to analysis. In *Compiler Construction*, pages 41–60. Springer, 2013.

[12] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, New York, NY, USA, 2013. ACM.

[13] O. Lhoták. *Program analysis using binary decision diagrams*. PhD thesis, McGill University, 2006.

[14] M. Might and O. Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. *ACM SIGPLAN Notices*, 41(9):13–25, 2006.

[15] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[17] M. Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications*, pages 189–234, 1981.

[18] O. Shivers. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM.

[19] J. Siek and W. Taha. Gradual typing for objects. *ECOOP '07: 21st European Conference on Object-Oriented Programming*, Jul 2007.

[20] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[21] H. E. van der Hoek. Object sensitive type analysis for PHP, 2013. `http://www.cs.uu.nl/people/jur/msctheses/henkerikvanderhoek-msc.pdf`.

[22] J. Verlaguet, J. Beales, E. Letuchy, G. Levi, J. Marcey, E. Meijer, A. Menghrajani, B. O'Sullivan, D. Paroski, J. Pearce, J. Pobar, and J. Van Dyke Watzman. The Hack language. `http://hacklang.org`, consulted Oct. 2014.

[23] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu. The HipHop compiler for PHP. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 575–586, New York, NY, USA, 2012. ACM.