Universiteit Utrecht

[Faculty of Science]
Information and
Computing Sciences

Utrecht University

Thesis

Object Sensitive Type Analysis for PHP

Henk Erik van der Hoek

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

Supervisors:
Utrecht University Dr. Jurriaan Hage
Utrecht University M.Sc. Ruud Koot

December 11, 2014

Contents

1 Introduction

In dynamically typed languages type checking is performed at run-time. Implementations keep track of a value's type by associating each value with a tag. Because type checking is deferred until run-time a type error like multiplying two Boolean values, is only discovered at run-time. At this point an implementation has two options: (1) it may silently coerce the value into a value of a suitable type, e.g. converting Boolean truth values into integer values or (2) it may throw a run-time exception. Most dynamically typed languages opt for the first option. PHP for example silently coerces any value into a value of a suitable type when needed, only resources like file handlers or database connections being an exception. This defeats the common good programming practice that errors should be caught as early in the development process as possible.

Statically typed languages on the other hand perform type checking at compile time, allowing type errors to be caught at the earliest possible stage. Static typing enables compilers to perform optimizations and ensures the absence of (certain classes of) type errors. However imposing a type system on a language has disadvantages as well. First the type checker runs at compile time and to do so it must be conservative and reject programs that may execute correctly. Second, programming in a language with explicit type checking is sometimes perceived as more difficult.

We shall try to overcome the problems of dynamically typed languages and gain the advantages of static typing by performing type inference at compile time. Ongoing research in this field led to a broad range of algorithms for many dynamically typed languages [16, 11, 6]. In this thesis we contribute the following to this field of research:

- We shall introduce the notion of an Extended Monotone Framework to cope with dynamically discovered call graph edges in a data flow analysis (see Section 4.5).
- We shall specify a type analysis for PHP as an instance of the Extended Monotone Framework (see Section 6.2).
- We shall specify several analysis variations, resulting in different object sensitive analyses. (see Section 8).
- We shall implement a prototype of the type analysis and experimentally evaluate its precision and performance for different analysis variations (see Section 9).

The result of our work may be beneficial to both IDE and compiler developers. Traditionally IDE's for PHP lack extensive support for features like on-the-fly auto-completion, documentation hits and type related error detection because type information is not available. In addition, type information may enable compiler engineers to perform various optimisation schemes. For example, type analysis plays an important role in HipHop, a PHP compiler developed by Facebook [28].

2 Related work

2.1 Type inference for dynamic languages

2.1.1 Javascript

Jensen, Møller and Thiemann [16] present 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 algorithm is implemented as a monotone framework instance with an elaborate lattice structure to model the heap. The analysis lattice is a tuple consisting of the computed call graph and an abstract state for each program point. The presence of first-class functions implies that the flow of abstract state information and the call graph are mutually dependent. Because of this, the call graph is constructed on the fly during fix point iteration. The necessary points-to information to construct call graph edges is computed as part of the type analysis.

The precision of the analysis is improved by employing a technique called recency abstraction. An analysis which uses recency abstraction keeps track of two abstractions for each allocation site: one for the most recently allocated object and another summary for all older objects. The most recent abstraction represents exactly one concrete object at run time. This enables the analysis to perform strong updates on this object, keeping the abstraction as precise as possible.

2.1.2 Python

Fritz [11] presents a static program analysis to infer type information for Python programs. Fritz focuses on balancing precision and cost with the aim to find an analysis which is both fast and precise enough to be used within interactive tools. The proposed analysis is based on data flow and is both flow and context sensitive. The analysis is capable of dealing with first class functions and Python's dynamic class system by adding control flow edges during the fix point iteration. The precision of the analysis is controlled by the parameters to the widening operator which control (1) the maximum size of a union type, (2) the maximum number of attributes of a class or instance and (3) the maximum nesting depth.

2.1.3 PHP

Camphuijsen [6] presents a typing 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 and polymorphic types for functions. The algorithm is implemented as a monotone framework instance. During the execution of each

transfer function constraints for the PHP expression at hand are generated, and the type of an expression is found by resolving these constraints. A widening operator is used to force termination in the presence of infinitely nested array structures. The type system supports polymorphic types for functions. However polymorphic type signatures are not inferred by the analysis but should instead be supplied by the end user. First-class functions and object oriented programming constructs are not supported by the analysis.

2.2 Heap Abstractions

In a program the heap can grow arbitrarily large. Any static program analysis should use some *finite* representation of the heap. We will discuss two different techniques to deal with heap-allocated data called shape analysis and recency abstraction.

2.2.1 Recency abstraction

Balakrishnan and Reps [2] present an abstraction for heap-allocated data called recency abstraction. The analysis keeps track of two abstractions for each allocation site l. One abstraction keeps track of the most recently allocated object at l and another abstraction keeps track of all older objects allocated at l [14]. Hence, the most recent abstraction represents precisely one concrete object at run-time. This enables the analysis to perform strong updates to any updated properties of this object.

2.2.2 Shape Analysis

Shape analysis [24] is more powerful than recency abstraction but it is also more resource-intensive to execute. The property space of the analysis consists of three parts: (1) an abstract state, which maps variables to abstract locations, (2) an abstract heap, which specifies links between abstract locations and (3) sharing information for the abstract locations.

2.3 Object sensitivity

Smaragdakis, Bravenboer and Lhoták [26] describe a framework in which it is possible to describe different variations of object sensitivity (see Section 4.4.2). Their abstract semantics are parametrized by two functions which manipulate contexts:

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

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 authors

show that it is possible to specify all past implementations of context sensitivity by choosing different *record* and *merge* functions. For example the original definition of an n-object sensitive analysis by Milanova et al [22]. is given by:

$$\begin{aligned} \mathbf{Context} &= \mathbf{Label}^n \\ \mathbf{HContext} &= \mathbf{Label}^n \\ record(l, \delta) &= cons(l, first_{n-1}(\delta)) \\ merge(l, \gamma, \delta) &= \gamma \end{aligned}$$

Smaragdakis et al. [26] also introduce the concept of type sensitivity as a approximation of object sensitivity. A type sensitive analysis is similar to an object sensitive analysis but instead of allocation site labels types are used as context elements. Their work shows that type sensitivity preserves much of the precision of object sensitivity at considerably lower cost.

3 The PHP Programming Language

PHP [1] is an open source general-purpose imperative object oriented scripting language primarily aimed at developing dynamic web pages. The language became popular in the late 90's and is still widely used today.

3.1 History

The first version of PHP was written by Rasmus Lerdorf in 1994. Initially PHP started as a set of Perl scripts to develop dynamic websites. During the next three years the language evolved and new features like built-in support for databases, cookies, and user defined functions where added. Around May 1998 PHP was installed on nearly 60.000 domains, which equated to approximately 1% of all Internet domains at that time. In 1998 the PHP interpreter was completely rewritten by Andi Gutmas and Seev Suraski. Their work added support for object oriented programming to the language [1].

3.2 Language features

This section gives an overview of the PHP programming language and in particular those features that are relevant for type analysis.

3.2.1 Type system

PHP does not require programmers to explicitly annotate a variable with a type declaration. Instead the type of a variable is determined by the context in which a variable is used. If a Boolean truth value is assigned to a variable $\$a$, the type of $\$a$ simply becomes a Boolean. If the next statement assigns an integer value to $\$a$, the type of $\$a$ becomes an integer. PHP will silently coerce any value into a value of a suitable type when needed, only resources like file handlers and database connections being an exception.

3.2.2 First class functions

PHP supports first class functions since version 5.3. PHP supports the passing of functions as parameters to other functions or methods, returning functions as values and assigning functions to variables.

```
1 $greet = function($name)
2 {
3     printf("Hello %s\r\n", $name);
4 };
5
6 $greet('World');
```

3.2.3 Class based object-oriented programming model

PHP supports class based object-oriented programming since version 3. PHP 5 introduced support for private and protected members and methods, abstract classes and methods, interfaces and exception handling. Every method call is resolved at run-time according to the actual type of the receiver object.

```
1  abstract class Vehicle {
2      abstract public function drive ();
3  }
4
5  class Car extends Vehicle {
6      protected var $colour;
7
8      function __construct ($colour)
9      {
10         $this->colour = $colour;
11     }
12
13     public function drive () {
14         echo "Drive the " . $this->colour . " car";
15     }
16 }
17
18 $x = new Car ("Red");
19 $x->drive ();
```

3.2.4 Type hinting

PHP supports type hinting since version 5. Type hinting allows formal parameters of functions and methods to be annotated with a type. The interpreter will force actual parameters to be instances of the declared type, otherwise a run-time exception is thrown.

```
1  class Car extends Vehicle {
2      protected var $engine;
3      protected var $colour;
4
5      function __construct (Engine $engine, $colour) {
6          $this->engine = $engine;
7          $this->colour = $colour;
8      }
9  }
```

3.2.5 References (Variable Aliases)

PHP offers two different ways to assign variables. First, by default variables are assigned by value. The entire value of the source expression is copied into the destination variable. As a consequence of these so called copy-on-assignment semantics the source and destination variables are independent. Changing one of these variables will have no effect on the other. Second, PHP also allows

the developer to assign values by reference. The destination variable becomes an alias for the source variable. Changing the destination variable modifies the source variable and vice versa. To implement this behaviour the PHP interpreter maintains a symbol table linking variable names to values [1]. The *unset* function is a special PHP function which removes a variable from the symbol table. The associated value is only cleaned up if the removed symbol was the last incoming pointer. To this end each value keeps track of its reference count. We shall illustrate the references by means of a small example:

Listing 1: An example to illustrate references

```
1 $a = 10;
2 $b = $a;
3 $c = &$b;
4 unset($b);
```

At the end of the program in Listing 1 the symbols and values related according to the schema in Figure 1

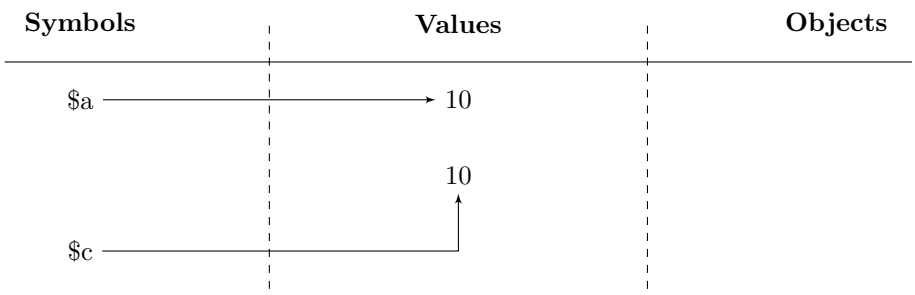

Figure 1: Schematic dipiction of symbols, values and objects for Listing 1

One should not confuse these variable aliases with a reference to heap allocated objects[2], these are two distinct concepts. To illustrate how references work in relation to heap allocated objects consider the example given in Listing 2:

Listing 2: References in relation to heap allocated objects

```
1 class X {
2     public $a = 1;
3     public $b = 2;
4 }
5
6 $x = new X ();
7 $y = &$x;
8 $r = &$x->b;
```

[1] In the PHP interpreter these values are known ZVals.
[2] By means of the *new* keyword

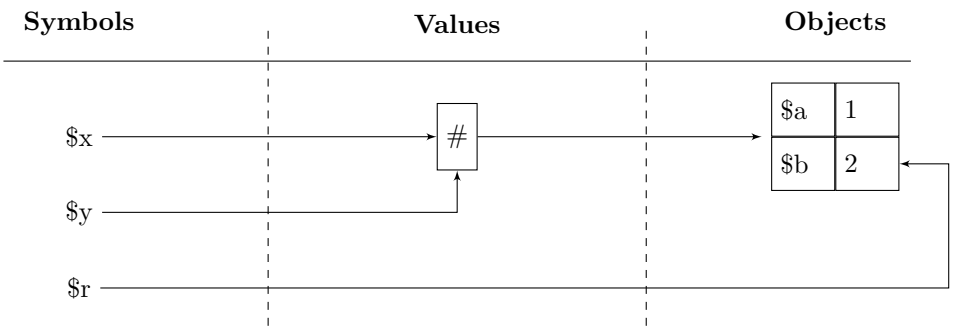

Figure 2: Schematic dipiction of symbols, values and object for Listing 2

4 Data Flow Analysis

Static program analysis offers compile-time techniques for predicting safe and computable approximations to the set of values that objects of a program may assume during its execution [24]. Because of the Halting problem precise answers are not computable and in the general case it is only possible to give approximate answers. Different approaches to program analysis are described in literature [8, 17, 23]. The type analysis described in this thesis is based on an approach called data flow analysis [17].

In data flow analysis a program is viewed as a graph where nodes represent elementary blocks and where edges indicate how control might pass between elementary blocks. The flow graph for the gcd program is shown in Figure 3.

$$\text{while } [k \neq m]^1 \text{ do if } [k > m]^2 \text{ then } [k = k - m]^3 \text{ else } [m = m - k]^4$$

An analysis is specified by a set of data flow equations. For each elementary block a pair of equations is defined. One equation in each pair specifies which information is true on the entry to a block, the second equation specifies which information is true at the exit of a block. A forward analysis propagates information in the direction of the control flow edges and a backwards analysis propagates information in the reverse direction.

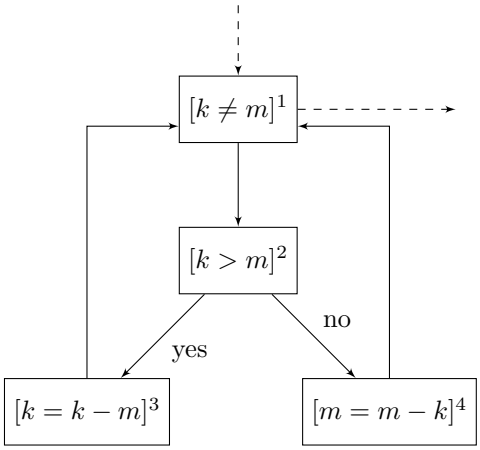

Figure 3: Flow graph for the greatest common divider program.

Throughout this document we will use P_* to denote the program which is currently being analyzed. $init$: **Program** $\rightarrow$ **Label** returns the initial label of a program, $final$: **Program** $\rightarrow \mathcal{P}$(**Label**) returns the set of final labels of a program, $labels$: **Program** $\rightarrow \mathcal{P}$(**Label**) returns the set of labels occurring in a program, $flow$: **Program** $\rightarrow \mathcal{P}$(**Label** $\times$ **Label**) returns the intra-procedural flow of a program and $flow^R$: **Program** $\rightarrow \mathcal{P}$(**Label** $\times$ **Label**) returns the reverse flow of a program. The reverse flow is defined by: $flow^R(P) = \{(l, l') | (l', l) \in flow(P)\}$.

4.1 Basic Definitions

A partially ordered set generalizes the concept of ordering the elements of a set. A partially ordered set $(L, \sqsubseteq)$ is a set L with a partial ordering $\sqsubseteq$ that indicates that one of the elements precedes the other for certain pairs of elements in the set. The relation $\sqsubseteq$ is partial to reflect the fact that there is no precedence relation between *every* pair of elements. In other words, it may be that for some pairs neither of the elements precedes the other.

Definition 1. *A partially ordered set $(L, \sqsubseteq)$ is a set L with a partial ordering $\sqsubseteq : L \times L \to \{\textbf{\textit{true}}, \textbf{\textit{false}}\}$. A partial ordering is a relation on a set L which is reflexive ($\forall l : l \sqsubseteq l$), anti-symmetric ($\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \implies l_1 = l_2$) and transitive ($\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \implies l_1 \sqsubseteq l_3$).*

Example: The power set $\mathcal{P}(L)$ is the set of all subsets of L, including the empty set $\emptyset$ and L itself. A partial ordering on $\mathcal{P}(L)$ is for example given by set inclusion. Formally: for every pair of subsets $S_1, S_2 \in \mathcal{P}(L)$:

$$S_1 \sqsubseteq S_2 \text{ iff } S_1 \subseteq S_2$$

A Hasse diagram is typically used to visualize a partially ordered set. For a partially order set $(L, \sqsubseteq)$ each element of L is represented as a vertex and an edge is drawn between x and y if x covers y. x covers y iff $x \sqsubset y$ and there is no z such that $x \sqsubset z \sqsubset y$, where $l \sqsubset l' = l \sqsubseteq l' \wedge l \neq l'$.

Example: The Hasse diagram of $(\mathcal{P}(\{x, y, z\}), \subseteq)$ is shown in Figure 4

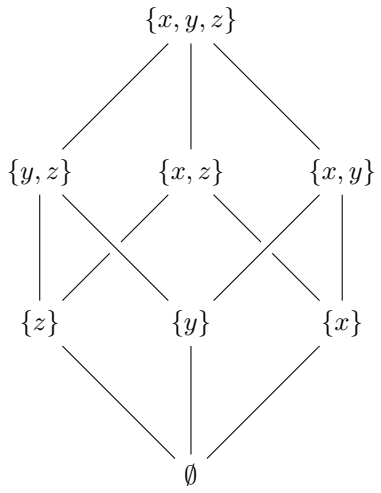

Figure 4: Hasse diagram of $(\mathcal{P}(\{x, y, z\}), \subseteq)$.

An element $l \in L$ is an upper bound of a subset Y of L if $\forall l' \in Y : l' \sqsubseteq l$. The unique least upper bound l of Y is an upper bound of Y for which holds that $l \sqsubseteq l_0$ for each other upper bound l_0 of Y. The least upper bound is denoted by the join operator $\bigsqcup : \mathcal{P}(L) \to L$ and often $l_1 \sqcup l_2$ is written as a shorthand for $\bigsqcup\{l_1, l_2\}$.

An element $l \in L$ is a lower bound of a subset Y of L if $\forall l' : l' \sqsupseteq l$. The greatest lower bound (denoted by the meet operator $\bigsqcap$) is defined analogously to the least upper bound.

Definition 2. *A complete lattice* $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ *is a partially ordered set such that all subsets have least upper bounds and greatest lower bounds.*

Example: Consider the power set $\mathcal{P}(L)$ with a partial ordering given by set inclusion (as in Figure 4). The least and greatest elements in $\mathcal{P}(L)$ are given by $\bot = \emptyset$ and $\top = L$ and the join and meet operators are given by:

$$S_1 \sqcup S_2 = S_1 \cup S_2$$
$$S_1 \sqcap S_2 = S_1 \cap S_2$$

So, the complete lattice in this example is given by $(\mathcal{P}(L), \subseteq, \cup, \cap, L, \emptyset)$.

A lattice L satisfies the ascending chain condition (ACC) if each ascending chain $(l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq ...)$ eventually stabilizes: $\exists n : l_n = l_{n+1}$. If a lattice has a finite height it satisfies the ascending chain condition.

4.2 Monotone Frameworks

A data flow analysis is specified by a set of equations. Each elementary block will give rise to a pair of equations: one equation will specify which information is true on the entry of the block and one equation will specify which information is true on the exit of a block:

$$\begin{aligned} \mathit{Analysis}_\circ(l) =& \bigsqcup \{\mathit{Analysis}_\bullet(l') \mid (l', l) \in F\} \sqcup \iota^l_E \\ & \text{where } \iota^l_E = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases} \\ \mathit{Analysis}_\bullet(l) =& f_l(\mathit{Analysis}_\circ(l)) \end{aligned} \tag{1}$$

A transfer function $f_l : L \to L$ for each $l \in \text{labels}(P_*)$ describes how information will flow from the entry to the exit of an elementary block. Increasing the information available at the entry of a block should naturally only increase (and not decrease) the information available at the exit of a block. Formally we will say the transfer functions are monotone, i.e. $l \sqsubseteq l' \implies f_l(l) \sqsubseteq f_l(l')$.

Instead of specifying the equations directly, we will introduce a framework which abstracts the commonalities and parametrizes the differences between different analyses. Identifying a framework makes it possible to design generic algorithms for equations solving, as we will show in Section 4.3. An instance of the frame-

work specifies a concrete analysis.

Definition 3. *An instance of a Monotone Framework* $(L, \mathcal{F}, F, E, \iota, \lambda l \rightarrow f_l)$ *consists of:*

- *A complete lattice* L *which satisfies the ascending chain condition.*
- *A set* $\mathcal{F}$ *of monotone transfer functions from* L *to* L *that contains the identity functions and that is closed under function composition.*
- *The finite flow* F
- *The finite set of extremal labels* E
- *The extremal value* $\iota \in L$
- *A mapping* $\lambda l \rightarrow f_l$ *from labels to transfer functions in* $\mathcal{F}$

A monotone framework instance gives rise to Equation 1. The analysis results are obtained by solving these equations.

4.3 The Worklist Algorithm

In Algorithm 1 we present a generic algorithm to compute the least solution to the data flow equations (see [24], Chapter 2.4).

The algorithm uses a worklist W to store a list consisting of tuples of program labels. We shall denote the first and the second component of a tuple of the worklist by l and l' respectively. The first component l of each tuple signifies a program label for which the data flow information has changed at its entry (in case of a forward analysis). We shall process the tuple in the worklist by propagating this change through the flow graph. First the transfer function is applied to the changed value. If the effect value is more informative than the entry value of l' the effect value is propagated in two steps: (1) the array A is updated and (2) the worklist W is updated by appending the outgoing edges of the node with program label l. Finally the analysis result is obtained by applying the transfer functions to the values in the array A.

The worklist algorithm will terminate if the following two sufficient conditions hold. First, the transfer functions are monotone and second, the lattice satisfies the ascending chain condition. It is possible to relax the last condition by employing a technique called widening [7]. However, widening is out of the scope of this thesis and we shall not rely on widening to guarantee termination.

Algorithm 1 Worklist Algorithm

Input: A Monotone Framework instance $(L, \mathcal{F}, F, E, \iota, \lambda l \rightarrow f_l)$
Output: $MFP_\circ$, $MFP_\bullet$

Step 1: Initialization

```
W ← nil
for (l, l') ∈ F do
    W ← cons ((l, l'), W)
for l ∈ F do
    if l ∈ E then
        A[l] ← ι
    else
        A[l] ← ⊥_L
```

Step 2: Iteration

```
while W ≠ nil do
    (l, l') ← head (W)
    W ← tail (W)
    if f_l (A[l]) ⋢ A[l'] then
        A[l'] ← A[l'] ⊔ f_l (A[l])
        for (l', l'') ∈ F do
            W ← cons ((l', l''), W)
```

Step 3: Presenting the results

```
for all l do
    MFP_∘(l) ← A[l]
    MFP_•(l) ← f_l (A[l])
```

4.4 Interprocedural Analysis

Almost any modern programming languages supports procedures or functions in some form. However, naively applying intraprocedural data flow techniques may harm the precision of the obtained results. If one procedure may be called from a dozen different locations the following questions arise: should one join the data flow information calculated for each calling location? And should one propagate a single result to each calling location at the end of a procedure?

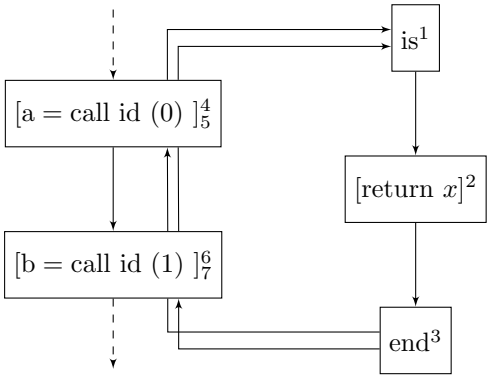

Figure 5: Flow graph for the example program.

We will illustrate by means of a small example that applying the intraprocedural data flow techniques naively leads to poisoning of the analysis result. Consider for example the example program shown below and its flow graph shown in Figure 5.

function id (x) is^1
  [return x]^2
end^3
[a = call id (0)]$^4_5$
[b = call id (1)]$^6_7$

To avoid poisoning, information should only flow along valid paths. A valid path starts at an extremal node and all procedure exits match the procedure entries, although it is possible that some procedures are only entered, and not yet exited. For example $[4, 1, 2, 3, 5]$ is a valid path while $[4, 1, 2, 3, 7]$ is not a valid path. Consider an analysis which determines the sign of every variable. The infinite domain of integers $\mathbb{Z}$ will be represented by $\mathcal{P}(\{\mathbf{-}, \mathbf{0}, +\})$. Clearly the abstract value for 0 and 1 are $\{\mathbf{0}\}$ and $\{+\}$. If information is allowed to flow only along valid paths, the analysis will conclude that the signs of a and b are indeed $\{\mathbf{0}\}$ and $\{+\}$ respectively. However, if information is also propagated along invalid paths the analysis result will be imprecise and conclude that the signs of a and b are both equal to $\{\mathbf{0}, +\}$.

The precision of an analysis is improved by separating data flow information depending on the calling context. The possibly infinite set of calling contexts is abstracted to a finite set of abstract contexts Δ. For each abstract context

$\delta \in \Delta$ we may have a different value for the original lattice L. So the complete lattice $\widehat{L}$ becomes:

$$\widehat{L} = \Delta \to L$$

The transfer functions associated with a procedure call and a procedure return handle the flow of information between different abstracts contexts in such a way that the information will flow along valid paths. So in a sense a context sensitive analysis simulates the behaviour of a call stack.

Choosing different context elements $\delta \in \Delta$ yields different context sensitive analyses. It is important to choose our abstract context wisely. Intuitively two opposing forces are at work if we increase the number of context elements. If we increase the depth of an analysis poisoning of the analysis results may be avoided. However, if poisoning does occur anyway, a deeper analysis will become less scalable since poisoning of the analysis results may quickly lead a combinatorial explosion of data flow facts being propagated. Intuitively, context partitions the data flow facts in different bins, if we increase the number of bins the redundancy should be minimal and the data flow facts should be separated in evenly distributed partitions. Hence, context elements should be as little correlated as possible for an analysis with high precision. [26]. We will describe two different flavours of context sensitivity, namely call-site sensitivity and object sensitivity.

4.4.1 Call site sensitivity

Employing call strings is the most common approach to context sensitivity. Call strings abstract the calling context into a list of procedure call labels seen during execution without seeing the corresponding return statement:

$$\Delta = \mathbf{Label}^*$$

During fixed point iteration new call strings are generated on-the-fly as call labels are added to the front of the list.

Example Consider a call to procedure f made from label 7 from a context [2]. The procedure f will be analyzed in the context $[7, 2]$.

Call strings may become infinitely long for recursive programs. As a consequence the lattice $\Delta \to L$ fails to satisfy the ascending chain condition (ACC) even though the lattice L might satisfy the ACC. Restricting the length of call strings to $\leq k$ is one way to enforce the ACC. Hence, by employing bounded call string we guarantee termination of the analysis at the price of lost precision.

Example: Let $k = 2$ and consider a call procedure f made from label 7 from a context $[7, 2]$. The procedure f will be analyzed under the context $[7, 7]$.

4.4.2 Object sensitivity

An object sensitive analysis uses object allocation site labels[3], instead of call site labels, as context elements. When a method is called the inferred facts are sepa-

[3]That is labels associated with the new statement

rated depending on the allocation site of the receiver object. For object oriented programs this approach gives a better precision compare to its costs [26].

```
1  class A {
2      function foo (Object $o) { ... }
3  }
4
5  class Client {
6      function bar(A $a1, A $a2) {
7          ...
8          $a1->foo($someobj1);
9          ...
10         $a2->foo($someobj2);
11         ...
12     }
13 }
```

In this example [26] an 1-object sensitive analysis will analyze foo separately depending on the allocation sites of the objects that $\$a1$ and $\$a2$ point to. It is not clear where the objects pointed to by $\$a1$ and $\$a2$ are allocated, so in this example it is not clear whether all calls to foo are handled under the same context or separately under different contexts.

Object sensitivity is particularly well suited context abstraction for analysing object oriented programs. However, a object sensitive analysis has many degrees of freedom relating to which context elements are selected upon each method invocation or object allocation [26]. So, in this thesis we shall build upon a framework by Smaragdakis et al. [26] to describe various forms of object sensitivity. Their framework offers a clean model to design and reason about different analysis variations. A analysis variations is given by defining the two context manipulation functions:

$$\begin{aligned} &\mathit{record} : \textbf{Label} \times \textbf{Context} \to \textbf{HContext} \\ &\mathit{merge} : \textbf{Label} \times \textbf{HContext} \times \textbf{Context} \to \textbf{Context} \end{aligned}$$

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 which we shall described in Section 6 uses these two context manipulation functions while different implementation, and hence different forms of object sensitivity, are given in Section 8.

4.5 Extended Monotone Frameworks

We will extend the notion of a Monotone Framework by taking context into account. Our definition of an Extended Monotone Framework is a generalization of an Embellished Monotone Framework as described by Nielson et al. [24] which enables us to add control flow edges dynamically. Our framework will abstract the commonalities and parametrize the differences between context sensitive analyses. On one side the framework is general enough to allow dynamic control

flow edges. But on the other side the framework is restrictive and forces us to implement a flow and context sensitive analysis.

We shall first redefine the notion of a program flow and introduce the concept of an inter-procedural flow. The program flow F defines how information flows between program points. A program point is a tuple (l, δ) consisting of a program label and a context element:

$$(l, \delta) \in \mathbf{Point} = \mathbf{Label} \times \mathbf{Context}$$

Hence the program flow states how information flows between elementary blocks *and* how information flows from one context to the other:

$$F \in \mathbf{Flow} = \mathcal{P}(\mathbf{Point} \times \mathbf{Point})$$

The interprocedural flow relates the program points of a procedure call to the corresponding program points of a procedure definition:

$$IF \in \mathbf{InterFlow} = \mathcal{P}(\mathbf{Point} \times \mathbf{Point} \times \mathbf{Point} \times \mathbf{Point})$$

We shall label the four individual components of interprocedural flow **c**all, e**n**try, e**x**it and **r**eturn:

$$((l_c, \delta_c), (l_n, \delta_n), (l_x, \delta_x), (l_r, \delta_r)) \in IF$$

We will use the interprocedural flow to let information flow only along valid paths. In general we shall enforce the following two constraints on the interprocedural flow:

Invariant 1. $\forall\ ((l_c, \delta_c), (l_n, \delta_n), (l_x, \delta_x), (l_r, \delta_r)) \in IF\ :\ (\ \delta_c = \delta_r \wedge \delta_n = \delta_x\ )$

Invariant 2. $\forall\ ((l_c, \delta_c), (l_n, \delta_n), (l_x, \delta_x), (l_r, \delta_r)), ((l'_c, \delta'_c), (l'_n, \delta'_n), (l'_x, \delta'_x), (l'_r, \delta'_r)) \in IF\ :\ (l_c = l'_c \iff l_r = l'_r)$

The first invariant states that the data flow information shall flow back to the original context after analyzing the callee and that the callee shall propagate the information from the entry to the exit under the same context. The second invariant states that there is a one-on-one relationship between a call label and its accompanying return label. The second invariant also allows us to see the interprocedural flow as a function from return label to call label: $IF(l_r)$ returns the corresponding call label. For convenience we shall also introduce the *returnPoints* function, which simply selects the fourth component of the interprocedural flow:

$$returnPoints(IF) = \{\ (l_r, \delta_r)\ |\ ((l_c, \delta_c), (l_n, \delta_n), (l_x, \delta_x), (l_r, \delta_r)) \in IF\ \}$$

Definition 4. *An instance of an Extended Monotone Framework is a 7-tuple* $(L, \mathcal{F}, E, \iota, \lambda l \to \lambda\delta \to f_{l,\delta}, \lambda l \to \lambda\delta \to \phi_{l,\delta}, \lambda l \to \lambda\delta \to next_{l,\delta})$ *and consists of:*

- *A complete lattice* L *which satisfies the ascending chain condition.*
- *A set* $\mathcal{F}$ *of monotone transfer functions from* L *to* L *that contains the identity function and that is closed under function composition.*
- *The finite set of extremal labels* E
- *The extremal value* $\iota \in L$
- *A mapping* $\lambda l \to \lambda\delta \to f_{l,\delta}$ *from labels and context elements to transfer functions in* $\mathcal{F}$
- *A mapping* $\lambda l \to \lambda\delta \to \phi_{l,\delta}$ *from labels and context elements to dynamic interprocedural flow creation functions in* L *to* $InterFlow$.
- *A mapping* $\lambda l \to \lambda\delta \to next_{l,\delta}$ *from labels and context elements to dynamic flow creation functions in* $InterFlow$ *to* $Flow$.

Notice that an instance of the Extended Monotone Framework does not require the complete program flow up front. Instead, an instance supplies the initial labels and a propagation function $next_{l,\delta}$. In the equations and the explanation below we shall concentrate on the case of a forward analysis. In case of a forward analysis, an instance of the Extended Monotone Framework gives rise to five data flow Equations (2) - (6):

$$\begin{aligned} A_\circ(l,\delta) = & \bigsqcup \{\, A_\bullet(l',\delta') \mid ((l',\delta'),(l,\delta)) \in F \,\} \sqcup \iota_E^{l,\delta} \\ & \text{where } \iota_E^{l,\delta} = \begin{cases} \iota & \text{if } l \in E \wedge \delta = \Lambda \\ \bot & \text{otherwise} \end{cases} \\ & \text{for all } (l,\delta) \text{ in } F \end{aligned} \tag{2}$$

Equation (2) specifies how information flows between program points. A program point is a tuple (l, δ) consisting of a program label and a context element. Hence, the first equation not only specifies how information flows between elementary blocks but also how information flows from one context to the other. The program flow F depends on Equation (5). This enables the analysis to add control flow edges dynamically.

$$\begin{aligned} A_\bullet(l,\delta) = & f_{l,\delta}\ (A_\circ(l,\delta)) \\ & \text{for all } (l,\delta) \text{ in } F \text{ except all } (l,\delta) \in returnPoints\ (IF) \end{aligned} \tag{3}$$

Equation (3) specifies how information flows from the entry to the exit of an elementary block. The transfer function $f_{l,\delta} : L \to L$ acts on the input based on the program label and the current analysis context. Access to the current analysis context is necessary in order to perform an object sensitive analysis: the transfer function for an object allocation makes a call to the *record* function which expects the current analysis context as one of its arguments.

$$\begin{aligned} A_\bullet(l_r,\delta_r) = & f_{l_c,l_r}\ (A_\circ(l_c,\delta_c), A_\circ(l_r,\delta_r)) \\ & \text{for all } (l_r,\delta_r) \in returnPoints\ (IF) \end{aligned} \tag{4}$$

Equation (4) specifies how information flows back from a procedure exit to the caller. The binary transfer function $f_{l_c,l_r} : L \times L \to L$ accepts two parameters. The first parameter describes the data flow information at the entry of a call and the second parameter describes the data flow information at the exit of the callee. The f_{l_c,l_r} function combines both lattice elements depending on the semantics of the language.

$$F = \bigcup (\{\ next_{e,\Lambda}\ (\emptyset) \mid e \in E\} \cup \{\ next_{l',\delta'}\ (IF) \mid ((l,\delta),(l',\delta')) \in F\ \}) \quad (5)$$

Equation (5) specifies how the program flow is given by first calculating the initial edges and expanding the flow graph by means of the $next_{l,\delta} : \textbf{InterFlow} \to \textbf{Flow}$ function until a fix point is reached.

$$\begin{aligned} IF = {} & \phi_{l,\delta}(A_\bullet(l,\delta)) \cup IF \\ & \text{for all } (l,\delta) \text{ in } F \end{aligned} \quad (6)$$

Equation 6 specifies how the $\phi_{l,\delta} : L \to \textbf{InterFlow}$ function adds elements to the inter procedural flow. The Equations (2) to (6) show that the program flow F, the inter procedural flow IF and the data flow information ($A_\circ$ and $A_\bullet$) are mutually dependent. The program flow F depends on the interprocedural flow IF (Equation 5), the interprocedural flow IF depends on the effect values $A_\bullet$ (Equation 6), the effect values $A_\bullet$ depend on the context values $A_\circ$ (Equation 3 and 4 while the context values depend on the program flow F (Equation 2). Hence the five equations are mutually dependent.

4.6 The Worklist Algorithm for the Extended Monotone Framework

In this section we shall present an iterative worklist algorithm for the Extended Monotone Framework. Given an instance of the Extended Monotone Framework Algorithm 2 computes the least fixed point solution.

The algorithm uses a worklist W to store a list consisting of tuples of program points [4]. We shall denote the first and the second component of a tuple of the worklist by (l,δ) and (l',δ') respectively. The first component (l,δ) of each tuple signifies a program point for which the data flow information has changed at its entry (in case of a forward analysis). We shall process the tuple in the worklist by propagating this change through the flow graph. First the transfer function is applied to the changed value. The unary transfer function is applied in all cases except if the program point (l,δ) corresponds to a method return block in which case the binary transfer function is applied. If the effect value is more informative than the entry value of (l',δ') the effect value is propagated in three steps: (1) the array A is updated, (2) the intraprocedural flow IF is updated by taking the union of the previously known intraprocedural flow and the result of the call $\phi_{l',\delta'}(A[(l',\delta')])$ and (3) the worklist W is updated by appending the result of the call $next_{l',\delta'}(IF)$. Finally, the analysis result is obtained by applying the transfer functions to the context values in the array A.

[4] The algorithm for the Extended Monotone Framework employs a worklist consisting of tuples of program points, in contrast to the original worklist algorithm which employs a worklist of program labels

Algorithm 2 Worklist Algorithm for the Extended Monotone Framework

Input: An Extended Monotone Framework instance $(L, \mathcal{F}, E, \iota, f_{l,\delta}, \phi_{l,\delta}, next_{l,\delta})$
Output: $MFP_\circ$, $MFP_\bullet$

Step 1: Initialization

IF $\leftarrow \emptyset$
W $\leftarrow nil$
for $l \in E$ **do**
  A$[l, \Lambda] \leftarrow \iota$
  for $((l,\delta),(l',\delta')) \in next_{l,\Lambda}(\emptyset)$ **do**
    W $\leftarrow$ cons $(((l,\delta),(l',\delta')), \text{W})$

Step 2: Iteration

while $W \neq nil$ **do**
  $((l,\delta),(l',\delta')) \leftarrow$ head (W)
  W $\leftarrow$ tail (W)

  if $(l,\delta) \in$ returnPoints (IF) **then**
    $l_c \leftarrow IF(l_r)$
    Effect $\leftarrow f_{l_c,l_r}$ (A$[l_c,\delta]$, A$[l,\delta]$)
  else
    Effect $\leftarrow f_{l,\delta}$ (A$[l,\delta]$)

  if Effect $\not\sqsubseteq$ A$[l',\delta']$ **then**
    A$[l',\delta'] \leftarrow$ A$[l',\delta'] \sqcup$ Effect
    IF $\leftarrow \phi_{l',\delta'}$ (A$[l',\delta']$) $\cup$ IF
    for $((l',\delta'),(l'',\delta'')) \in next_{l',\delta'}$ (IF) **do**
      W $\leftarrow$ cons $(((l',\delta'),(l'',\delta''), \text{W})$

Step 3: Presenting the results

for all l and δ **do**
  $MFP_\circ(l,\delta) \leftarrow$ A$[l,\delta]$
  if $(l,\delta) \in$ returnPoints (IF) **then**
    $l_c \leftarrow IF(l_r)$
    $MFP_\bullet(l,\delta) \leftarrow f_{l_c,l_r}$ (A$[l_c,\delta]$, A$[l,\delta]$)
  else
    $MFP_\bullet(l,\delta) \leftarrow f_{l,\delta}$ (A$[l,\delta]$)

Note that Algorithm 2 is slightly different in case of a backwards analysis. A binary transfer function will be used to merge two lattice elements at each program point signifying a procedure call (not a procedure return)

5 Control Flow Graphs for PHP

5.1 Representations

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 are lowered to a sequence of simple statements. Furthermore, since class and function identifiers are case insensitive in PHP all class and function identifiers are rewritten to their lowercase form. Except rewriting, the translation to the intermediate representation also takes care of including files[5]. However these simplifications do not change the expressiveness of the language. A subset of the IR which we shall call the core IR will be specified in a formal manner in Section 5.2. The core IR includes language features like class definitions, object allocation, method invocation and field reads and writes. A formal description of our type analysis on the core IR is given in Section 6 while additional language feature like native functions, exceptions and arrays are explained in Section 7. Figure 6 displays the relationship between PHP and the two intermediate representations.

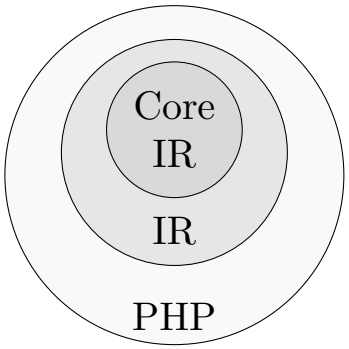

Figure 6: Relationship between PHP and the two intermediate representations.

[5]Only if it is possible to statically determine the file name of the included file, dynamically included files are not supported.

5.2 Core Intermediate Representation

5.2.1 Abstract Grammar

Grammar 1 describes the core representation. In the core representation each elementary block is annotated with a distinct label.

---

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

P ::= C S

C ::= **class** c M | **class** c **extends** c M | C C

M ::= [**function** m $(\vec{p})]^{l_n}_{l_x}$ S | M M

S ::= [v = C]^l
| [v = v]^l
| [v = v op v]^l
| S_1 **;** S_2
| [**skip**]^l
| **if** [v]^l **then** S_1 **else** S_2
| **while (true)** S
| [**break**]^l
| [**continue**]^l
| [**new** c]^l
| [v.f = v]^l
| [v = v.f]^l
| [v = v.m $(\vec{p})]^{l_c}_{l_r}$
| [**return** v]^l

C ::= **true** | **false** | **null** | n

Grammar 1: The core IR grammar

5.2.2 Functions

By structural induction we shall describe several functions which operate on the program under analysis, where the program is represented using the Core IR:

▶ $init$: **Program** $\rightarrow \mathcal{P}$(**Label**)
The $init$ function returns the initial labels of a program. For a forward analysis, the initial labels corresponds to the elementary block in which the execution of a program will start.

▶ $flow$: **Program** $\rightarrow$ **Flow**
The $flow$ function returns the intraprocedural flow of a program. Due to dynamic dispatch the interprocedural flow shall not be part of the flow returned by this function. Instead the interprocedural flow will be obtained on the fly during fixed point iteration.

▶ $entry$: **Program** $\times$ **Method** $\rightarrow$ **Label**
The $entry$ function returns the entry label l_n of a method. A method is a tuple consisting of a class name and a method name, or formally: **Method** = **ClassName** $\times$ **MethodName**.

▶ $exit$: **Program** $\times$ **Method** $\rightarrow$ **Label**
The $exit$ function returns the exit label l_x of a method.

▶ $return$: **Program** $\times$ **Label** $\rightarrow$ **Label**
The $return$ function returns the return label l_r corresponding to a given call label l_c.

▶ $resolve$: **Program** $\times$ **Method** $\rightarrow$ **Method**
The $resolve$ function resolves a dynamic method call by traversing the inheritance hierarchy. We shall explain this function by means of an example. Consider the PHP program given in 3 where the type analysis has determined that the variable x on line 15 refers to an object of type $Child$. A call will be made to the $resolve$ function with as a first parameter the program under analysis and as a second parameter the tuple $(Child, foo)$. By traversing the inheritance hierarchy the $resolve$ will establish that the method being called is $(Parent, foo)$.

Listing 3: Resolving method calls

```
1  abstract class Parent {
2      public function foo () {
3          ...
4      }
5  }
6
7  class Child extends Parent {}
8
9  $x = new Child ();
10 $x->foo ();
```

▶ $className$: **Program** $\times$ **Label** $\rightarrow$ **ClassName**
The $className$ function returns the class name of the allocated object given an allocation site label.

Throughout this document we shall denote the program under analysis as P_*. As a notational shortcut we may omit the first parameter of the functions described

above by annotating the function name with a star. For example, $init_*$ simply returns the initial label of the current program under analysis.

6 Type Analysis

In this section we shall explain the type analysis operating on the core intermediate representation (see Section 5.2). The *Type Analysis* will determine:

> For each program point, which types may a variable assume at the exit from the point.

Before giving a formal specification of the analysis, lets look at an example. The PHP program in Listing 4 constructs and evaluates an arithmetic expression using a simple grammar consisting only of the terminal symbol $Number$ and the non-terminal symbol $Multiply$.

Listing 4: PHP program of an expression evaluator lowered to the core IR

```
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
13     $r = $this->r;
14     $y = $r->evaluate ();
15
16     $z = $x * $y;
17     return $z;
18   }
19 }
20
21 $x = new Value ();
22 $v = 10;
23 $x->v = $v;
24
25 $y = new Value ();
26 $v = false;
27 $y->v = $v;
28
29 $z = new Multiply ();
30 $z->l = $x;
31 $z->r = $y;
32
33 $r = $z->evaluate ();
```

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 14 is a Boolean. Note that in order to determine the receiver method of the method call on line 11, line 14 and line 33 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.

In order to full fill these requirements the type analysis shall extend the notion of a points-to analysis. A points-to analysis computes a static approximation of all the heap objects that a pointer variable can reference during 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 during run-time. This approach is similar to the approach taken by Jensen, Møller and Thiemann [16] to perform a type analysis on Javascript programs.

In Section 6.1 we shall describe how an abstraction of a run-time value will be represented using a lattice. The presence of dynamic method dispatching in PHP implies that the flow of abstract state information and call graph information are mutually dependent. Our Extended Monotone Framework (see Section 4.5) captures this behaviour intuitively. In Section 6.2 we shall formulate the type analysis as an instance of the Extended Monotone Framework.

6.1 The Analysis Lattice

We shall model an abstract value by a tuple. Each component of the tuple contains an abstraction for a specific type: integers, Booleans, objects, etc. One may view our type analysis as an extension of a points-to analysis since one of the tuple components contains the points-to information.

Abstract Signs. The abstract representation of integer values is given by its sign set. Formally a numeric value is abstracted to an element of the set $\mathcal{P}(\textbf{Sign})$ where **Sign** is given by:

$$sign \in \textbf{Sign}$$
$$sign ::= \texttt{-} \mid \texttt{0} \mid \texttt{+}$$

The elements in the power set $\mathcal{P}(\textbf{Sign})$ form a lattice by inclusion. For convenience, we shall define a function, $lift$, which accepts a non deterministic binary operator and lifts it to accept sets of values as it arguments:

$$lift : (\alpha \times \beta \to \mathcal{P}(\gamma)) \to (\mathcal{P}(\alpha) \times \mathcal{P}(\beta) \to \mathcal{P}(\gamma))$$
$$lift(op) = \lambda as \to \lambda bs \to \bigcup \{\, op(a, b) \mid a \in as,\ b \in bs \,\}$$

We shall use this function to extend the notion of the usual arithmetic operators to the domain of signs. For example the addition operator on signs is defined by:

$$
\begin{aligned}
&\hat{+} : \mathcal{P}(\textbf{Sign}) \times \mathcal{P}(\textbf{Sign}) \rightarrow \mathcal{P}(\textbf{Sign}) \\
&\hat{+} = \textbf{let}\ f(\texttt{+}\ ,\ \texttt{+}) = \{\texttt{+}\} \\
&\qquad\qquad f(\texttt{+}\ ,\ \texttt{0}) = \{\texttt{+}\} \\
&\qquad\qquad f(\texttt{+}\ ,\ \texttt{-}) = \{\texttt{-}, \texttt{0}, \texttt{+}\} \\
&\qquad\qquad f(\texttt{0}\ ,\ \texttt{+}) = \{\texttt{+}\} \\
&\qquad\qquad f(\texttt{0}\ ,\ \texttt{0}) = \{\texttt{0}\} \\
&\qquad\qquad f(\texttt{0}\ ,\ \texttt{-}) = \{\texttt{-}\} \\
&\qquad\qquad f(\texttt{-}\ ,\ \texttt{+}) = \{\texttt{-}, \texttt{0}, \texttt{+}\} \\
&\qquad\qquad f(\texttt{-}\ ,\ \texttt{0}) = \{\texttt{-}\} \\
&\qquad\qquad f(\texttt{-}\ ,\ \texttt{-}) = \{\texttt{-}\} \\
&\quad \textbf{in}\ lift\ (f)
\end{aligned}
$$

Finally we define a function to obtain a sign given a concrete numeric value:

$$
\begin{aligned}
&fromInteger : \textbf{N} \rightarrow \textbf{Sign} \\
&fromInteger(n) = \begin{cases} \texttt{-} & \textbf{if}\ n < 0 \\ \texttt{0} & \textbf{if}\ n = 0 \\ \texttt{+} & \textbf{if}\ n > 0 \end{cases}
\end{aligned}
$$

Abstract Booleans. The abstract Booleans are elements of the set $\mathcal{P}(\textbf{Bool})$ where **Bool** is given by:

$$
\begin{aligned}
&b \in \textbf{Bool} \\
&b ::= \texttt{true} \mid \texttt{false}
\end{aligned}
$$

Once again, the elements in the power set $\mathcal{P}(\textbf{Bool})$ form a lattice by inclusion. Similarly to arithmetic operators, we shall lift the logical operators to the abstract domain of Boolean sets. For example the lifted *xor* operator is defined as:

$$
\begin{aligned}
&\widehat{xor} : \mathcal{P}(\textbf{Bool}) \times \mathcal{P}(\textbf{Bool}) \rightarrow \mathcal{P}(\textbf{Bool}) \\
&\widehat{xor} = lift(\lambda a \rightarrow \lambda b \rightarrow \{a\ xor\ b\})
\end{aligned}
$$

Abstract Addresses. 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 4.4.2). The analysis will keep track of an abstract object for each heap context. So we shall use heap context elements as abstract addresses. Formally, abstract addresses are elements of the set $\mathcal{P}(\textbf{HContext})$ where **HContext** depends on the chosen analysis variation (see Section 8). The elements in the power set $\mathcal{P}(\textbf{HContext})$ form a lattice by inclusion.

The definition of **HContext** depends on the chosen analysis variations but we shall require the existence of a complete function $label : \textbf{HContext} \rightarrow \textbf{Label}$ from heap context to allocation site label. In theory, this additional constraint reduces the degrees of freedom one may have in choosing the *record*

and *merge* functions. However, in practice this constraint does not prevent us from specifying all of the common analysis variations.

Abstract Values. An abstract value is represented using a tuple where each component describes a different type of value. We will define **Null** to be equal to $\{\top, \bot\}$. Formally the abstract value lattice is the Cartesian product of its components:

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

Example. Assume that allocation site labels are used as a heap context elements. An abstract value of $(\{[7]\}, \bot, \bot, \top)$ then signifies that the concrete value may either be an abstract object with a heap context of [7] or the *null* value.

We define an auxiliary function which injects individual components, like a sign set, into the abstract value lattice.

$$\begin{aligned}
&inject_{\mathcal{P}(\mathbf{HContext})} : \mathcal{P}(\mathbf{HContext}) \to \mathbf{Value} \\
&inject_{\mathcal{P}(\mathbf{HContext})}\ (l) = (l, \bot, \bot, \bot)
\end{aligned}$$

$$\begin{aligned}
&inject_{\mathcal{P}(\mathbf{Bool})} : \mathcal{P}(\mathbf{Bool}) \to \mathbf{Value} \\
&inject_{\mathcal{P}(\mathbf{Bool})}\ (l) = (\bot, l, \bot, \bot)
\end{aligned}$$

$$\begin{aligned}
&inject_{\mathcal{P}(\mathbf{Sign})} : \mathcal{P}(\mathbf{Sign}) \to \mathbf{Value} \\
&inject_{\mathcal{P}(\mathbf{Sign})}\ (l) = (\bot, \bot, l, \bot)
\end{aligned}$$

$$\begin{aligned}
&inject_{\mathbf{Null}} : \mathbf{Null} \to \mathbf{Value} \\
&inject_{\mathbf{Null}}\ (l) = (\bot, \bot, \bot, l)
\end{aligned}$$

PHP silently coerces any value into a value of a suitable type when needed. The type analysis needs to mimic this behaviour. Therefore we shall define a *coerce* function. The *coerce* function takes an abstract value and coerces it into a value of one of its components:

$$\begin{aligned}
&coerce_{\mathcal{P}(\mathbf{HContext})} : \mathbf{Value} \to \mathcal{P}(\mathbf{HContext}) \\
&coerce_{\mathcal{P}(\mathbf{HContext})}\ (\gamma s, bs, ss, n) = \gamma s
\end{aligned}$$

$$\begin{aligned}
&coerce_{\mathcal{P}(\mathbf{Bool})} : \mathbf{Value} \to \mathcal{P}(\mathbf{Bool}) \\
&coerce_{\mathcal{P}(\mathbf{Bool})}\ (\gamma s, bs, ss, n) = \\
&\quad \mathbf{let}\ \ signToBool\ (\texttt{-}) = \texttt{true} \\
&\qquad\qquad signToBool\ (\texttt{0}) = \texttt{false} \\
&\qquad\qquad signToBool\ (\texttt{+}) = \texttt{true} \\
&\qquad\qquad nullToBool\ (\top) = \{\texttt{false}\} \\
&\qquad\qquad nullToBool\ (\bot) = \emptyset \\
&\quad \mathbf{in}\ \{\ \texttt{true} \mid \gamma \in \gamma s\ \} \cup bs \cup \{\ signToBool(s) \mid s \in ss\ \} \cup nullToBool(u)
\end{aligned}$$

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

$$
\begin{array}{l}
coerce_{\mathbf{Null}} : \mathbf{Value} \to \mathbf{Null} \\
coerce_{\mathbf{Null}}\ (\gamma s, bs, ss, n) = n
\end{array}
$$

Example Consider the abstract value $v = (\bot, \bot, \{\texttt{+}\}, \bot)$ which signifies a positive integer value. The $coerce_{\mathcal{P}(\mathbf{Bool})}$ function coerces this abstract value into an abstract Boolean set, i.e. $coerce_{\mathcal{P}(\mathbf{Bool})}(v)$ equals $\{\texttt{true}\}$.

Using the *inject* and *coerce* functions we are able to lift operators like $\hat{+}$ and $\widehat{xor}$ to abstract values. The typical pattern is to coerce the arguments, apply the unlifted operator and inject the result back into an abstract value. For example, the lifted $\tilde{+}$ and $\widetilde{xor}$ operators are defined by:

$$
\begin{array}{l}
\tilde{+} : \mathbf{Value} \times \mathbf{Value} \to \mathbf{Value} \\
a\ \tilde{+}\ b = inject_{\mathcal{P}(\mathbf{Sign})}\ (coerce_{\mathcal{P}(\mathbf{Sign})}\ (a)\ \hat{+}\ coerce_{\mathcal{P}(\mathbf{Sign})}\ (b))
\end{array}
$$

$$
\begin{array}{l}
\widetilde{xor} : \mathbf{Value} \times \mathbf{Value} \to \mathbf{Value} \\
a\ \widetilde{xor}\ b = inject_{\mathcal{P}(\mathbf{Bool})}\ (coerce_{\mathcal{P}(\mathbf{Bool})}\ (a)\ \widehat{xor}\ coerce_{\mathcal{P}(\mathbf{Bool})}\ (b))
\end{array}
$$

In the end a type analysis should infer a type set for each variable, not an abstract value. However it is straight forward to obtain a type set given an abstract value. We shall view a type to be the set of class names augmented with primitive types. Formally we shall write:

$$\tau \in \mathbf{Type} = \mathbf{ClassName}_* + \{\ \text{Integer},\ \text{Boolean},\ \text{Null}\ \}$$

Now we are ready to define a function from an abstract value to a type set. Note that due to the dynamic nature of PHP it is not always possible to infer a single type. Consequently, the *type* function results a type set and not a single type. The $className_*$ and *label* functions are used to translate a heap context to a class name. The $className_*$ function is given in Section 5.2.2 and the *label*

function depends on the chosen analysis variation and is given in Section 8.

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

Example Consider the abstract value $v = (\bot, \{\texttt{true}\}, \{\texttt{+}\}, \bot)$ 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 }.

Abstract states. The analysis will operate on abstract states consisting of an abstract stack component and an abstract heap component. The abstract stack maps identifiers to abstract values while the abstract heap maps a tuple of a heap context and a field name to abstract values.

First we shall define the abstract stack. An identifier may be an ordinary variable, a method parameter or one of the two special identifiers. The two special identifiers ($\mathbb{R}$, $\mathbb{T}$) are used to signify the return value of a method invocation and the *this* variable which references the receiver object.

$$z \in \mathbf{Ident} = \mathbf{Var}_* + \mathbf{Z} + \{\mathbb{R}, \mathbb{T}\}$$

The parameter identifiers are modelled by $\mathbf{Z}$. On each method invocation the actual parameters are translated to numeric parameter identifiers based on their position in the source code. Subsequently on each method entry the numeric parameter identifiers are translated to formal parameters. Since $\mathbf{Var}_*$ is finite and the number of parameters in a method call is finite it is clear that **Ident** is always finite. Now we can view the abstract stack as an element of:

$$
\begin{array}{l}
S \in \mathbf{Stack} = \mathbf{Ident}\ \mapsto\ \mathbf{Value} \\
S ::= [\,] \mid S[z\ \mapsto\ v]
\end{array}
$$

Formally, S is a list but we will treat the abstract stack as a finite mapping. Thus, we write $dom(S)$ for $\{z \mid S\ contains [z \mapsto ...]\}$. If $z \in dom(S)$ we shall write $S(z) = v$ for the rightmost occurrence of z in S.

The other component of the abstract state is the abstract heap H. The abstract heap specifies a set of links between a heap context and an abstract value. Multiple links are distinguished by a field name.

$$
\begin{array}{l}
H \in \mathbf{Heap} = (\mathbf{HContext}\ \times\ \mathbf{FieldName}_*)\ \mapsto\ \mathbf{Value} \\
H ::= [\,] \mid H[(\gamma, f) \mapsto v]
\end{array}
$$

Again, we shall assume the usual notational shortcuts and view $H(\gamma, f) = v$ as the rightmost occurrence of (γ, f) in H. We call a tuple consisting of an

abstract stack and an abstract heap the abstract state. Formally we write:

$$\sigma \in \mathbf{State} = \mathbf{Stack} \times \mathbf{Heap}$$

We shall extend the **State** with a least element $\bot$ to capture the case that an elementary block is not reachable, for which we write $\mathbf{State}_\bot$. The partial ordering $\sqsubseteq$ on $\mathbf{State}_\bot$ is defined by:

$$\forall \sigma \in \mathbf{State}_\bot : \bot \sqsubseteq \sigma$$

The transfer functions shall operate on abstract states using nine state manipulation functions which act as an interface. The first five (*empty*, *read*, *write*, *readField* and *writeField*) are used for the intraprocedural fragment of the transfer function while the last four (*toParameters*, *toVariables*, *clearStack* and *clearHeap*) are used for the interprocedural fragment of the transfer function. The *empty* function simply returns a state consisting of an empty stack and an empty heap:

$$\begin{aligned} &\mathit{empty} : \mathbf{State} \\ &\mathit{empty} = ([], []) \end{aligned}$$

The *read* function reads an abstract value from the abstract stack given an identifier. If the identifier is not in the domain of the abstract stack, we shall return an abstract *null* value instead.

$$\begin{aligned} &\mathit{read} : \mathbf{Ident} \times \mathbf{State} \to \mathbf{Value} \\ &\mathit{read}\ (z, (S, H)) = \begin{cases} S(z) & \textbf{if } z \in \mathit{dom}(S) \\ \mathit{inject}_{\mathbf{Null}}\ (\top) & \textbf{otherwise} \end{cases} \end{aligned}$$

The *write* function takes an identifier and an abstract value and creates a binding on the abstract stack:

$$\begin{aligned} &\mathit{write} : \mathbf{Ident} \times \mathbf{Value} \times \mathbf{State} \to \mathbf{State} \\ &\mathit{write}\ (z, v, (S, H)) = (S[z \mapsto v], H) \end{aligned}$$

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:

$$\begin{aligned} &\mathit{readField} : \mathbf{Ident} \times \mathbf{FieldName} \times \mathbf{State} \to \mathbf{Value} \\ &\mathit{readField}\ (z, f, (S, H)) = \\ &\quad \textbf{let } \gamma s = \mathit{coerce}_{\mathcal{P}(\mathbf{HContext})}\ (\mathit{read}\ (z, (S, H))) \\ &\quad \textbf{in } \begin{cases} \bigsqcup \{\ \mathit{readHeap}\ (\gamma, f, H) \mid \gamma \in \gamma s\} & \textbf{if } \gamma s \neq \emptyset \\ \mathit{inject}_{\mathbf{Null}}\ (\top) & \textbf{otherwise} \end{cases} \end{aligned}$$

The *readHeap* function reads an abstract value from the heap given a heap context, γ, and a field name, f. If the tuple (γ, f) is not in the domain of the abstract heap, the abstract *null* value is returned instead:

$$readHeap : \mathbf{HContext} \times \mathbf{FieldName} \times \mathbf{Heap} \to \mathbf{Value}$$
$$readHeap\ (\gamma, f, H) = \begin{cases} H((\gamma, f)) & \textbf{if } (\gamma, f) \in dom(H) \\ inject_{\mathbf{Null}}\ (\top) & \textbf{otherwise} \end{cases}$$

The *writeField* function updates the abstract state to reflect a field assignment. First the abstract value corresponding to the identifier z is read from the stack. This abstract value may refer to multiple heap context elements. For each heap context element the abstract heap is updated:

$$writeField : \mathbf{Ident} \times \mathbf{FieldName} \times \mathbf{Value} \times \mathbf{State} \to \mathbf{State}$$
$$writeField\ (z, f, v, (S, H)) =$$
$$\begin{array}{lll} \textbf{let } \gamma s & = & coerce_{\mathcal{P}(\mathbf{HContext})}\ (read\ (z, (S, H))) \\ \quad H' & = & \bigsqcup \{ writeHeap(\gamma, f, v, H) \mid \gamma \in \gamma s \}) \\ \textbf{in } (S, H') & & \end{array}$$

The auxiliary *writeHeap* writes an abstract value to the heap. If the heap already contains a value for the given pair (γ, f) the previous and the new value are joined together:

$$writeHeap : \mathbf{HContext} \times \mathbf{FieldName} \times \mathbf{Value} \times \mathbf{Heap} \to \mathbf{Heap}$$
$$writeHeap\ (\gamma, f, v, H) = \begin{cases} H[(\gamma, f) \mapsto H((\gamma, f)) \sqcup v] & \textbf{if } (\gamma, f) \in dom(H) \\ H[(\gamma, f) \mapsto v] & \textbf{otherwise} \end{cases}$$

The next four functions are used to implement the interprocedural fragment of the transfer function. Parameter passing is implemented by means of the *toParameters* and *toVariables* functions and the *clearStack* and *clearHeap* function are used to implement the return of a procedure call.

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.

$$toParameters : \mathcal{P}(\mathbf{Var}_*) \times \mathbf{State} \to \mathbf{State}$$
$$toParameters\ (\vec{p}, (S, H)) = (\bigsqcup \{\ [i \mapsto S(p_i)] \mid p_i \in \vec{p}\ \}, H)$$

The *toVariables* function translates the parameter position 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.

$$toVariables : \mathcal{P}(\mathbf{Var}_*) \times \mathbf{State} \to \mathbf{State}$$
$$toVariables\ (\vec{p}, (S, H)) = (\bigsqcup \{\ [p_i \mapsto S(i)] \mid p_i \in \vec{p}\ \} \sqcup [\mathbb{T} \mapsto S(\mathbb{T})], H)$$

The *clearStack* and the *clearHeap* functions clear the stack and heap compo-

nents of the abstract state:

$$clearStack : \mathbf{State} \rightarrow \mathbf{State}$$
$$clearStack\ ((S, H)) = (\bot, H)$$

$$clearHeap : \mathbf{State} \rightarrow \mathbf{State}$$
$$clearHeap\ ((S, H)) = (S, \bot)$$

6.2 The Analysis

We shall specify the analysis as an instance of the Extended Monotone Framework $(\mathbf{State}_\bot, \mathcal{F}_{State}, init(P_*), \iota^{TA}, f^{TA}_{l,\delta}, \phi^{TA}_{l,\delta}, next^{TA}_{l,\delta})$. This instance gives rise to the following set of equations:

$$A_\circ(l,\delta) = \bigsqcup \{\ A_\bullet(l',\delta') \mid ((l',\delta'),(l,\delta)) \in F\ \} \sqcup \iota^{l,\delta}_E$$
$$\text{where } \iota^{l,\delta}_E = \begin{cases} \iota^{TA} & \text{if } l \in E \wedge \delta = \Lambda \\ \bot & \text{otherwise} \end{cases}$$
$$\text{for all } (l,\delta) \text{ in } F$$

$$A_\bullet(l,\delta) = f^{TA}_{l,\delta}\ (A_\circ(l,\delta))$$
$$\text{for all } (l,\delta) \text{ in } F \text{ except all } (l_r,\delta_r) \text{ in } IF$$

$$A_\bullet(l_r,\delta_r) = f^{TA}_{l_c,l_r}\ (A_\circ(l_c,\delta_c), A_\circ(l_r,\delta_r))$$
$$\text{for all } (l_r,\delta_r) \text{ in } IF$$

$$F = \bigcup \{\ next^{TA}_{e,\Lambda}\ (\emptyset) \mid e \in E\} \cup \{\ next^{TA}_{l',\delta'}\ (IF) \mid ((l,\delta),(l',\delta')) \in F\ \}$$

$$IF = \phi^{TA}_{l,\delta}(A_\bullet(l,\delta)) \cup IF$$
$$\text{for all } (l,\delta) \text{ in } F$$

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

6.2.1 The Extremal Value ι^{TA}

The extremal value ι^{TA} specifies the initial analysis information. Initially, we create an empty abstract state. This indicates that initially the extremal labels are reachable.

$$\iota^{TA} : \mathbf{State}_\bot$$
$$\iota^{TA} = empty$$

6.2.2 The $f^{TA}_{l,\delta}$ Function

The intraprocedural fragment. The transfer function $f^{TA}_{l,\delta} : \mathbf{State}_\perp \to \mathbf{State}_\perp$ specifies how flow and context sensitive type information flows from the entry to the exit of an elementary block. The behaviour of the transfer function is conditional, the data flow information should only be propagated if the elementary block is reachable. A elementary block is reachable if the abstract state is unequal to the least element, hence:

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

If the elementary block is reachable the transfer function delegates control to the $\psi_{l,\delta} : \mathbf{State} \to \mathbf{State}$ function. The $\psi_{l,\delta}$ function is restricted to only accept and return reachable states. Its implementation depends on the various forms of elementary blocks which we shall discuss next.

Transfer function for $[skip]^l$ and $[b]^l$. Boolean tests and skip nodes do not modify the abstract state, hence we simply specify $\psi_{l,\delta}$ to be the identify function.

$$\psi_{l,\delta}\ (\sigma) = \sigma$$

Transfer function for $[v = b]^l$ where b is a Boolean value. Assigning a Boolean value to a variable results in converting the Boolean value into an abstract value. A binding between this abstract value and the identifier v is then created in the abstract state, σ.

$$\begin{aligned} \psi_{l,\delta}\ (\sigma) = \mathbf{let}\ & value = inject_{\mathcal{P}(\mathbf{Bool})}\ (\{b\}) \\ \mathbf{in}\ & write\ (v, value, \sigma) \end{aligned}$$

Transfer function for $[v = n]^l$ where n is an integer value. Assigning an integer value to a variable results in converting the integer value into an abstract value using the $inject_{\mathcal{P}(\mathbf{Sign})}$ and $fromInteger$ functions:

$$\begin{aligned} \psi_{l,\delta}\ (\sigma) = \mathbf{let}\ & value = inject_{\mathcal{P}(\mathbf{Sign})}\ (fromInteger\ (n)) \\ \mathbf{in}\ & write\ (v, value, \sigma) \end{aligned}$$

Transfer function for $[v = v']^l$ where v' is a variable. An assignment will be processed by reading the abstract value for the identifier v' and writing it to v:

$$\begin{aligned} \psi_{l,\delta}\ (\sigma) = \mathbf{let}\ & value = read\ (v', \sigma) \\ \mathbf{in}\ & write\ (v, value, \sigma) \end{aligned}$$

Transfer function for $[v = v'\ \odot\ v'']^l$**.** The transfer function for a binary operator first reads the arguments v' and v'' from the abstract state σ. The resulting two abstract values are then applied to the lifted operator $\widetilde{\odot}$. Finally, the result is written back to the abstract state.

$$\begin{aligned} \psi_{l,\delta}\ (\sigma) = \mathbf{let}\ & value = read\ (v', \sigma)\ \widetilde{\odot}\ read\ (v'', \sigma) \\ \mathbf{in}\ & write\ (v, value, \sigma) \end{aligned}$$

Transfer function for $[v = \texttt{new } C]^l$. The transfer function for an object allocation uses the *record* function to create a new heap context, γ. The value of γ depends on the allocation site label l and the current analysis context δ. We can obtain different implementations of an object sensitive analysis by choosing different *record* and *merge* functions. We will describe several variations in Section 8.

$$\begin{aligned}\psi_{l,\delta}\ (\sigma) = \textbf{let}\ \ \gamma &= record\ (l, \delta)\\ value &= inject_{\mathcal{P}(\textbf{HContext})}\ (\{\gamma\})\\ \textbf{in}\ write\ &(v, value, \sigma)\end{aligned}$$

Transfer function for $[v.f = v']^l$. A value is written to a field by reading the abstract value corresponding to the identifier v' from the abstract state σ. A binding between this abstract value and the identifier v and selector f is then created in the abstract state:

$$\begin{aligned}\psi_{l,\delta}\ (\sigma) = \textbf{let}\ \ value &= read(v', \sigma)\\ \textbf{in}\ writeField\ &(v, f, value, \sigma)\end{aligned}$$

Transfer function for $[v = v'.f]^l$. A value is read from a field by reading the abstract value corresponding to $v'.f$ and writing it to the identifier v:

$$\begin{aligned}\psi_{l,\delta}\ (\sigma) = \textbf{let}\ \ value &= readField(v', f, \sigma)\\ \textbf{in}\ write\ &(v, value, \sigma)\end{aligned}$$

The interprocedural fragment. Only the intraprocedural transfer functions remain to be specified. These transfer function specify how the abstract state information flows between method calls.

Transfer function for $[v = v'.method\ (\vec{p})]^{l_c}$. The transfer function for a method call 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.

$$\begin{aligned}\psi_{l_c,\delta}\ (\sigma) = \textbf{let}\ \ value &= read(v', \sigma)\\ \sigma' &= toParameters\ (\vec{p}, \sigma)\\ \textbf{in}\ write&(\mathbb{T}, value, \sigma')\end{aligned}$$

Transfer function for $[C.method(\vec{p})]^{l_n}$. The transfer function for a method entry implements the second half of the parameter passing semantics by translating the parameter positions to the formal parameter identifiers using the *toVariables* function.

$$\psi_{l_n,\delta}\ (\sigma) = toVariables\ (\vec{p}, \sigma)$$

Transfer function for $[\texttt{return } v]^l$. The transfer function for return statements creates a binding between the special return identifier $\mathbb{R}$ and the abstract value

corresponding to the identifier v. The transfer function for a method return will use the special return identifier again to retrieve the return value.

$$\begin{aligned}\psi_{l,\delta}\ (\sigma) = \textbf{let}\ \ value\ \ &=\ \ read(v, \sigma)\\ \textbf{in}\ write\ (\mathbb{R}, value, \sigma)\end{aligned}$$

Transfer function for $[C.method(\vec{p})]^{l_x}$. A method exit does not modify the abstract state, hence we specify $f^{TA}_{l_x}$ to be the identify function:

$$\psi_{l_x,\delta}\ (\sigma) = \sigma$$

Transfer function for $[v = v'.method\ (\vec{p})]^{l_r}$. A binary transfer function is used to propagate information to the exit of a method return block. The first parameter of $f^{TA}_{l_c,l_r} : \textbf{State}_\perp \to \textbf{State}_\perp \to \textbf{State}_\perp$ describes the data flow information at the entry of the method call and the second parameter describes the data flow information at the entry of the method return, which equals to the information at the exit of the method body. Similarly to the unary transfer function, we only wish to propagate the data flow information if both the method call and the method return elementary blocks are reachable:

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

The $\psi_{l_r,l_r} : \textbf{State} \to \textbf{State} \to \textbf{State}$ is a binary function operating on abstract states. It propagates the stack information from the first parameter and the heap information and the return value from the second parameter:

$$\begin{aligned}\psi_{l_c,l_r}\ (\sigma, \sigma') = \textbf{let}\ \ value\ \ &=\ \ read\ (\mathbb{R}, \sigma')\\ \sigma''\ \ &=\ \ clearHeap\ (\sigma)\ \sqcup\ clearStack\ (\sigma')\\ \textbf{in}\ write\ (v, value, \sigma'')\end{aligned}$$

6.2.3 The $\phi^{TA}_{l,\delta}$ Function

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

The $\phi^{TA}_{l_c,\delta}$ function proceeds by retrieving 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 run-time, 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 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:

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

The *edge* function depends on a couple of previously defined functions. In Section 6.1 we required the existence of a complete function *label* to map a heap context to an allocation site label. The $className_*$, $resolve_*$, $entry_*$, $exit_*$ and $return_*$ functions give information about the program under analysis and are defined in Section 5.2.2.

Dynamic interprocedural flow function for any other elementary block. Any other elementary block does not add interprocedural flow edges, hence we simply specify $\phi^{TA}_{l\delta}$ to return the empty set.

$$\phi^{TA}_{l,\delta}\ (\sigma) = \emptyset$$

6.2.4 The $next^{TA}_{l,\delta}$ Function

The $next^{TA}_{l,\delta}$ function enables the Extended Monotone Framework to incorporate control flow edges which are due to the dynamically discovered call edges. The $next^{TA}_{l,\delta}$ function will behave differently depending on the elementary block that corresponds to l. Three different cases are distinguished: (1) a method call, (2) a method return and (3) any other elementary block.

Next function for $[v = v'.method\ (\vec{p})]^{l_c}$. 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. Simultaneously, information needs to propagate back from the callee to the caller:

$$
\begin{array}{ll}
next_{l_c,\delta}\ (IF) & = \{((l_c, \delta), (l_n, \delta'))\ |\ ((l_c, \delta), (l_n, \delta'), (l_x, \delta'), (l_r, \delta)) \in IF\ \} \\
& \cup \{((l_x, \delta'), (l_r, \delta))\ |\ ((l_c, \delta), (l_n, \delta'), (l_x, \delta'), (l_r, \delta)) \in IF\ \}
\end{array}
$$

It is necessary to add the return edge immediately since the propagation from the entry to the exit of the callee may hold if no new information becomes available. We shall illustrate this by means of an example. Consider the program given below:

function id (x) is^1
  [return x]^2
end^3
[a = call id (true)]$^4_5$
[b = call id (true)]$^6_7$

The *id* function is called twice with identical parameters. On the first call, the data flow information will be propagated through the function body until the

end elementary block and back to the caller. However, on the second call no new information becomes available and hence the data flow propagation will hold and not flow back to the caller if the return edge is not added immediately.

Next function for $[end^{l_x}]$. 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:

$$next_{l_x,\delta}\ (IF) = \{\ ((l_x, \delta), (l_r, \delta')) \mid ((l_c, \delta'), (l_n, \delta), (l_x, \delta), (l_r, \delta')) \in IF\ \}$$

Example Consider a method call $[a = b.foo()]^{l_c}_{l_r}$, which is being analyzed under context δ_a, resulting in the method $X.foo$ being analyzed under δ_b. At the end of the method body of $X.foo$ the information will be propagated to l_r under the original context δ_a.

Next function for any other elementary block. If l corresponds to any other elementary block the information will be propagated in the usual manner. The information will flow to the adjacent elementary blocks as given by $flow_*$ under the same context:

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

Note that the usage of $flow_*$, and not $flow_*^R$, implies that the type analysis is a forward analysis.

7 Additional Language Features

In this section we shall briefly describe additional language features supported by our type analysis.

7.1 Resources, String, Doubles and Arrays

In Section 6.1 we kept the definition on an abstract value deliberately incomplete. Space constraints prevented us from including other value types like strings, doubles, resources and arrays. We shall briefly explain how these values types are implemented. Strings are represented using a lattice which either $\bot$, some concrete value or $\top$. The Hasse diagram is shown below:

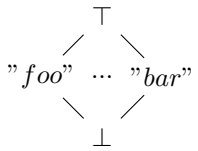

Figure 7: Hasse diagram of the string lattice

Doubles are represented using their sign set analogous to integer. Resources are a value type specific for PHP. They are commonly used as a reference to a file or a database connection. The set of distinct resource types is finite so it is a natural choice to model resources as a power set lattice. Modelling arrays is slightly more complicated. To this end the heap lattice is redefined as:

$$H \in \mathbf{Heap} = (\mathbf{HContext} \times \mathbf{Index}) \mapsto \mathbf{Value}$$

Where an index is either a field name or one of the two special index fields $\mathbb{K}$ and $\mathbb{V}$ which represent the array key and value respectively:

$$\mathbf{Index} = \mathbf{FieldName}_* + \{\mathbb{K}, \mathbb{V}\}$$

An advantage of this approach is the possibility to model infinite array structures. Furthermore, the coercion rules are extended to include these additional value types.

7.2 Native Constants, Functions and Classes

The PHP programming language comes with a standard library of native function and classes. In addition, the PHP language lends its popularity to the numerous extensions[6] which allows a PHP developer to quickly glue various software systems together. So, support for native functions and classes is necessary to support any real world PHP application. Our type analysis deals with native functions and classes by assigning a type signature to them. We shall illustrate the semantics of our type signatures by means of an example. The

[6] http://www.php.net/manual/en/extensions.membership.php

$fgetc$ function reads a single character from a file. A file resource is expected as the first and only argument. The return value is either a string containing the read character or the Boolean value false if the EOF [7] is encountered. The type signature reflects this:

$$\text{fgetc} :: \{resource(file)\} \rightarrow \{string, false\}$$

Furthermore, the type signatures are flexible enough to deal with parametric polymorphism. The $array\_search$ function for example returns the key for a needle if it is found in the input array:

$$\text{array\_search} :: \{\beta\} \rightarrow \{array(\{\alpha\} \Rightarrow \{\beta\})\} \rightarrow \{boolean\}? \rightarrow \{\alpha, false\}$$

In PHP an array key may either be an integer or a string. The type signature of the $array\_search$ function expects a needle of type β and an array with a key of type α and a value of type β and returns the key of type α corresponding to the needle if present in the array or $false$ otherwise. The third parameter of the $array\_search$ function is optional. This is reflected in the type signature by a question mark.

Native classes are handled by assigning a class signature to each native class. A class signature consists of three sections to define constants, public and protected fields and public and protected methods. We shall illustrate the semantics of a class signature by means of an example

ziparchive = {
  [*constants*]
  OVERWRITE :: *integer*
  CREATE :: *integer*
  ...

  [*fields*]
  status :: *integer*
  comment :: *string*
  ...

  [*methods*]
  addfile :: {*string*} → {*string*}? → {*integer*}? → {*integer*}? → {*boolean*}
  close :: {*boolean*}
  open :: {*string*} → {*integer*}? → {*integer*, *boolean*}
  ...
}

Method signatures are similar to function signatures.

[7] End Of File

7.3 Exceptions

Exception analysis is mutually recursive with the points-to fragment of our type analysis: handling exceptions causes exceptional control flow which makes more code reachable while points to information determines which objects are thrown at a *throw* statement. Different approaches to exception analysis are described in literature. Some rely on a conservative approximation of exception handling [20, 19] by modelling exception throwing as an assignment to one single global variable for every thrown exception in the program. This special variable is then subsequently read at each catch clause. Although sound this approach is highly imprecise. Precise exception handling is implemented in the *DOOP* framework [5]. The exception analysis logic fully models the Java semantics for exceptions, Java being their language under analysis. In this scheme imprecision is only introduced due to the static approximation of objects in the underlining points-to analysis. However due to the precise representation of exception objects this approach has a higher space and time cost. Exception handling in our type analysis takes the same approach as the *DOOP* framework.

7.4 Abstract Garbage Collection

The idea of abstract garbage collection [21] is similar to its concrete counterpart. Concrete garbage collection reallocates memory as fresh if a heap allocated object is not referenced any longer, either by another object on the heap or a variable on the stack. Abstract garbage collection however may work a bit more aggressively by reallocating a heap allocated abstract object as fresh if it is not referenced by another object on the heap or by a (stack) variable currently in scope. It may seem counter intuitive to be more aggressive in the abstract garbage collector. However, this is possible because the analysis maintains an abstract representation of the heap for each program point where the program at run-time only maintains one concrete heap.

Abstract garbage collection is performed by the transfer function responsible for handling a method entry. So in contrast to concrete garbage collection, which may happen at any point while executing the program, abstract garbage collection is only performed at a specific point.

8 Analysis Variations

In Section 6 we have specified the type analysis but until now we have refrained from specifying the *record*, *merge* and *label* functions. In this section we shall define multiple definitions of the *record*, *merge* and *label* functions which characterize different variations of an object sensitive analysis. This approach is due to Smaragdakis et al. [26] and we shall use their terminology to discuss the following kinds of sensitivity: full-object, plain-object and type sensitivity.

8.1 Full-Object Sensitivity

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 allocation site labels, these labels are effectually used to split the data flow facts to differentiate between multiple method invocations. Milanova [22] 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:

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

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.

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).

Naming conventions. We shall introduce an abbreviation for 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 shall write $n$full+mH. This off-by-one notation may seem odd, but it is self-evident that at least one allocation site label will be used to represent an abstraction of a heap object. Since only context depths of two and lower are within bounds of present day technology the following analysis variations are of particular interest: 1full, 1full+1H and 2full+1H.

The *label* function is not a context manipulation function but rather a necessary artefact of our type analysis which requires the existence of a complete function from heap context to allocation site label.

8.2 Plain-Object Sensitivity

An object sensitive analysis uses a heap context as an abstraction for an object. In a full-object sensitive analysis the context under which a method will be analyzed depends on the **full** heap context of the receiver object. In contrast, a plain-object sensitive analysis combines both the heap context of the receiver object and the regular context of the caller:

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

Both full-object and plain-object sensitive analysis 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 [19] is an example of a framework which uses plain-object sensitivity.

Analysis. We shall compare a full-object with a plain-object sensitive analysis, both of context depth two. On one hand a full-object sensitive analysis will analyse each method using as a context the allocation site label of the receiver object and the allocation site label of the receiver object allocator. On the other hand a plain-object sensitive analysis will analyze each method using as a context the allocation site label of the receiver object and the allocation site label of the caller object. But which approach gives the best precision? Theoretically one may argue that a full-object sensitive analysis will outperform a plain-object sensitive analysis. To avoid poisoning context elements should be as little correlated as possible. However when merging the allocation site label of the receiver object with the allocation site label of the caller object the two labels are likely to be correlated. For example the receiver and the caller object are exactly the same if an object calls a method on itself. Based on this observation we expect a full-object sensitive analysis to outperform a plain-object sensitive analysis of the same context depth in terms of precision.

Naming conventions. 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.

8.3 Type Sensitivity

As stated in Section 4.4 the precision of an analysis is improved by separating data flow information depending on the calling context. We abstracted the set of possibly infinite calling contexts to a finite set of abstract contexts δ. As specified above, the abstract contexts in an object sensitive analysis are lists of allocation site labels. Since a typical program has many allocation sites this quickly leads to a combinatorial explosion of abstract contexts. The idea behind 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 smaller than the number of allocation sites. For a 2type+1H analysis we shall define the following context manipulation functions:

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

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. We shall refrain from specifying the auxiliary function $\mathcal{T} : \mathbf{Label} \to \mathbf{ClassName}$ until the next section.

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

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

This choice of context is interesting, because we expect the number of context elements to be smaller compared to number of context elements in a 2full+1H analysis, but greater than the number of context elements in a 2type+1H analysis.

8.3.1 Choice of type

In Section 4.4 we explained, as a rule of thumb, that context elements should be as little correlated as possible. In this section we shall discuss which type

will constitute a good choice with this rule in mind. Consider an allocation statement $[obj = new\ A]^l$ inside a class C. Given the label l the function $\mathcal{T}$ may return:

- The dynamic type A of the *allocated* object
- An upper bound C on the dynamic type of the *allocator* object. We can only establish an upper bound on the dynamic type because a subclass may choose not to override the method containing the allocation site.

We shall explain why returning the dynamic type A of the allocated object is not a good design choice. For context to do its work well, the elements should be as little correlated to each other as possible. One may view flow sensitivity as an instance of context sensitivity where the analysis computes a result for each program point (l, δ). Hence, to obtain a scalable analysis the elements of the context δ should be as little correlated to the program label l. However, the dynamic type A of the allocated object is closely related to the program label l: the fact that we are analyzing a method, let say $X :: foo$, already gives us an upper bound on the dynamic type of the receiver object: it should either be X or any subclass which does not override the foo method.

So the function $\mathcal{T}$ should return the upper bound C on the dynamic type of the allocator object. This type is less correlated to the program label l and hence improves the scalability of our analysis.

9 Experimental Evaluation

9.1 Setting

We have implemented and evaluated several of the analysis variations, as described in Section 8, up to a context depth of 2. In this Section we aim to answer the following questions:

- Is our implementation sound with respect to the observed types during sample runs of a test suite?
- Does full object sensitivity achieve a better precision than plain object sensitivity for a context depth of 2?
- Does type sensitivity achieve a better performance than object sensitivity while maintaining most of its precision?
- Does enabling abstract garbage collection result in a better performance?

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.

9.1.1 Implementation

The implementation consists of two distinct phases. In the first phase, an intermediate representation is obtained by parsing the original PHP program using PHC [3]. PHC is a framework to parse PHP programs to abstract syntax trees. In a pipeline of sequential transformations the original AST is lowered to various intermediate forms. We lower the original PHP program 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. The source code is freely available on: http://www.github.com/henkerik/objectsensitivetyping/

9.1.2 Test suite

The PHP programs in the test suite are shown in Table 1. It was necessary to make small modifications to the original programs on some occasions due to the use of unsupported language features. These modifications are documented in a file called modifications.txt, which is present in the directory of each project. A list of unsupported PHP features is given in Appendix A.

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 [4].	486
MVC	A framework which implements the model-view-controller pattern for web application.	2583
Dijkstra	An implementation of Dijkstra's algorithm [9] using adjacency lists to represent a graph structure.	4854
Floyd	An implementation of the Floyd-Warshall algorithm [10] using an adjacency matrix to represent a graph structure.	5742
Interpreter	An object oriented implementation of a small expression language, including a parser.	843

Table 1: List of projects in the test suite

9.2 Result

9.2.1 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 the *type* function (see Section 6.1).

The run-time type sets are obtained by instrumenting the original source code of the programs listed in Table 1. 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 means of the *get_class* and the *get_resource_type* functions respectively. This results in a run-time type set since each assignment may be executed multiple times with possibly different values, and hence different types, being assigned.

We compared the observed run-time types with the inferred type sets by our implementation. On all assignments the observed type sets are a subset of the

inferred type set. In other words, the type analysis was able to infer all types observed at run-time. Furthermore, we analyzed on how many occasions the inferred type set precisely matches the observed type set. The results for this experiment are shown in Table 2:

	insensitive	1obj	1obj+1H	2plain+1H	2full+1H
raytracer	360	81	0	6	12
gaufrette	432	4	-4	0	0
phpgeo	518	20	0	0	0
mime	211	0	2	0	0
mvc	173	40	8	0	0
dijkstra	321	2	0	0	0
floyd	556	1	2	0	0
interpreter	265	2	0	0	0

Table 2: Number of precise matches for each analysis variations

9.2.2 Comparing plain and full-object sensitivity

We shall compare the precision of plain-object sensitivity to full-object sensitivity. Based upon our theoretical discussion of plain and full object sensitivity (see Section 8.2) 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. Additionally, 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 [8] 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:

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

		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

Table 3: Comparison of plain and full object sensitivity

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

Table 4: Collapsing types

Table 3 shows the precision metrics for 2plain+1H and 2full+1H analysis variations for our test suite. All metrics 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. Further, the first four metrics are given relative to the immediately preceding column, only the metrics in the insensitive column are absolute numbers.

Discussion. In terms of precision, the 2plain+1H and 2full+1H analysis variations show exactly the same result for 7 out of the 8 test programs. Only for the `raytracer` program the 2full+1H analysis achieves a better precision. So, although the difference between both analysis variations is minimal, the only difference which we observed confirms the theoretical merits of full object sensitivity. The difference between the precision scores is minimal, we suspect that this is due to the relative small size of the test programs.

In terms of performance, the 2full+1H analysis 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 is 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. These results clearly show that there exists no trade-off between precision and performance. On the contrary, the higher precision enables the analysis to exclude a broader range of target methods while resolving a method call.

9.2.3 Comparing type sensitivity and object sensitivity

Next we compare the performance of type sensitivity to object sensitivity. Based upon our theoretical discussion of type sensitivity we expect that type sensitivity achieves a better performance than object sensitivity while maintaining most of its precision (see Section 8.3). The results of our experiments are shown in Table 5.

Discussion. In terms of performance, the 2type+1H analysis only outperforms the 2full+1H analysis for 3 of the 8 test programs. Compared to the 2full+1H analysis, the 1type1obj+1H analysis does not perform better for a single test program. So in contradiction with 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

		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

Table 5: Comparison of type sensitivity and object sensitivity

by two opposing forces. On 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 to the implementation used by Smaragdakis et al. [26], 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 done by Smaragdakis et al. [26] using their implementation.

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, the 2type+1H analysis performs worse than the 2full+1for 3 out of the 8 test programs. For the 1type1obj+1H analysis we observe identical results for 7 out of the 8 test programs. Only for the *raytracer* test program the 1type1obj+1H analysis performs worse in terms of precision.

9.2.4 Abstract Garbage Collection

Our experiments show that the type analysis only terminates within a reasonable amount of time if abstract garbage collection (see Section 7.4) is enabled. Abstract garbage collection prevents the propagation of abstract objects which are known to be unreachable. Table 6 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

Table 6: Performance Metrics of Abstract Garbage Collection

10 Conclusion

In this thesis 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. To this end we extended the notion of a Monotone Framework. Our Extended Monotone Framework (Section 4.5) intuitively captures the notion of a dynamically discovered call graph and enables us to add control flow edges on the fly. An instance of the Extended Monotone Framework gives rise to a set of mutually recursive equations. These equations define the propagated data flow information, the program flow and the interprocedural flow in terms of each other. A worklist algorithm (see Section 4.6) was given to compute the least fixed point solution.

We specified the type analysis as an extension of a points-to analysis expressed as an instance of the Extended Monotone Framework (see Section 6). In addition, we presented a novel method to capture the coercion rules of PHP by means of the *coerce* and *reject* functions. The transfer functions rely on work by Smaragdakis et al. [26] by employing the context manipulation functions *record* and *merge* to capture the essence of object sensitivity. Multiple variations of an object sensitive analysis are obtained by choosing different implementations for these context manipulation functions. In Section 8 we discussed several interesting choices.

Our experimental evaluation (see Section 9) aims to answer four questions. First, is our implementation sound? Since PHP lacks a formal specification we established the soundness of our analysis by comparing the inferred types to the types observed at run time. Second, does full-object sensitivity achieve a better precision than plain-object sensitivity? Our experiments show similar precision metrics for both plain and full-object sensitivity for most of the test programs. However for the only program where a difference is observed, the full-object sensitive analysis indeed achieved an increase in precision. Third, does type sensitivity achieve a better performance than object sensitivity while maintaining most of its precision? Our experiments show that type sensitivity does not result in improve in performance, on the contrary: a the 2full+1H analysis out performs the 2type+1H analysis on 5 of the 8 test programs, while the 1type1obj+1H analysis is slower for all test programs. Four, does enabling abstract garbage collection result in a better performance? Our experiments do indeed clearly show an improvement in performance if abstract garbage collection is enabled.

11 Future Work

Our work leaves many open questions and future research areas. To start, our implementation does not support all of PHP's language features. A list of unsupported PHP features is given in Appendix A. Adding features like first class function and closures requires an even more elaborate lattice to model the abstract state. Similar to dynamic dispatch first class function require on-the-fly call graph creation. We expect than one may benefit from our Extended Monotone Framework to cope with the additional dynamic flow edges.

Another interesting line of research is whether employing recency abstraction improves the precision of the analysis. Recency abstraction allows strong updates on fields in certain cases. Jensen, Møller and Thiemann [16] describe a type analysis of Javascript programs and they observed an improvement in precision due to recency abstraction.

Kastrinis et al. [18] propose significant optimizations to the exception handling mechanism. They observed 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 observation gave them the suggestion to coarsen the representation of an exception in two different ways: (1) exception objects are always handled in an context insensitive manner and (2) exception objects are merged and represented as one abstract object per dynamic type. Kastrinis et al. [18] show that the performance of an analysis is significantly improved by coarsening the representation of exceptions objects. It would be interesting to investigate whether our type analysis benefits from coarsening of exception objects as well.

Our type analysis is path insensitive although there are strong reasons to believe that adding path sensitivity may be a beneficial technique for improving the precision of our type analysis. Path sensitivity answer the question question whether a given path in the control flow edge is executable [13]. A common approach in PHP is to accept parameters of a mixed type and apply conversions on them:

```
1  <?php
2  function wrapper ($input)
3  {
4      if (!is_array ($input))
5          $input = array ($input);
6
7      return processArray ($input);
8  }
9  $x = wrapper (1);
10 $y = wrapper (array (2));
11 ?>
```

On line 7 a path sensitive analysis will conclude that the $*input* variable is always an array of integers. However, a path insensitive analysis will falsely conclude that the $*input* variable is either an integer or an array of integers.

Finally, work by Smaragdakis and Bravenboer [27] points to a further interesting area of research. Their implementation of a points-to analysis is based on a framework called *DOOP* [5]. *DOOP* uses Datalog to define analyses and

heavy optimizations are performed on a Datalog level. It would be interesting to investigate if their declarative approach extends to our type analysis. For example, it remains unclear to us if Datalog is expressive enough to describe the coercion rules of PHP.

References

[1] Zend php, January 2014.

[2] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. *Static Analysis*, pages 221–239, 2006.

[3] Paul Biggar, Edsko de Vries, and David 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.

[4] 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.

[5] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Notices*, volume 44, pages 243–262. ACM, 2009.

[6] Patrick Camphuijsen. Soft typing and analyses on php programs, 2007.

[7] Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

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

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

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

[11] Levin Fritz. Balancing cost and precision of approximate type inference in python, 2011.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 52–58. ACM, 2005.

[14] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. *ECOOP 2010–Object-Oriented Programming*, pages 200–224, 2010.

[15] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[16] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[17] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

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

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

[20] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Compiler Construction*, pages 153–169. Springer, 2003.

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

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

[23] F. Nielson and H. Nielson. Type and effect systems. *Correct System Design*, pages 114–136, 1999.

[24] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[25] Mooly Sagiv, Thomas Repst, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating, 1996.

[26] Yannis Smaragdakis and Martin Bravenboer. Pick your contexts well: Understanding object-sensitivity the making of a precise and scalable pointer analysis.

[27] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis.

[28] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen 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.

Appendices

A List of Unsupported Features

Our type analysis does **not** support the following PHP features:

- **String coercion.** The magic method *__toString* is called if an object is coerced to a string (for example by using the *strval* function.)
- **Object cloning.** An object copy is created by using the *clone* keyword. When an object is cloned, a shallow copy will be created.
- **Namespaces.** Class definitions may be partitioned in different namespaces.
- **Array of characters.** A string maybe treated as an array of characters. Individual characters may be indexed using the ussual array indexing notation.
- **Anonymous functions.** PHP allows the creation of anonymous function which may be stored in variables, passed as arguments or used as the return value of a function call.
- **References.** References (see Section 3.2.5) are meant to access a variables content by different names. References add an additional level of indirection and should not be confused with a variable referencing heap allocated data.
- **SPL.** The OO standard library of PHP is not fully supported. In particular the *ArrayAccess*, *Clonable*, and *Iterator* interfaces are not supported.
- **Eval.** The *eval* function is not supported by the type analysis.
- **Object destruction.** Destructors are not supported by the type analysis.

B Control Flow of Example Program

The flow diagram in Figure 8 shows the intraprocedural and interprocedural flow of the example program in Listing 4 using normal and thick lines respectively. The interprocedural flow is discovered on the fly while performing the type analysis as shown in the iteration steps given in Appendix C.

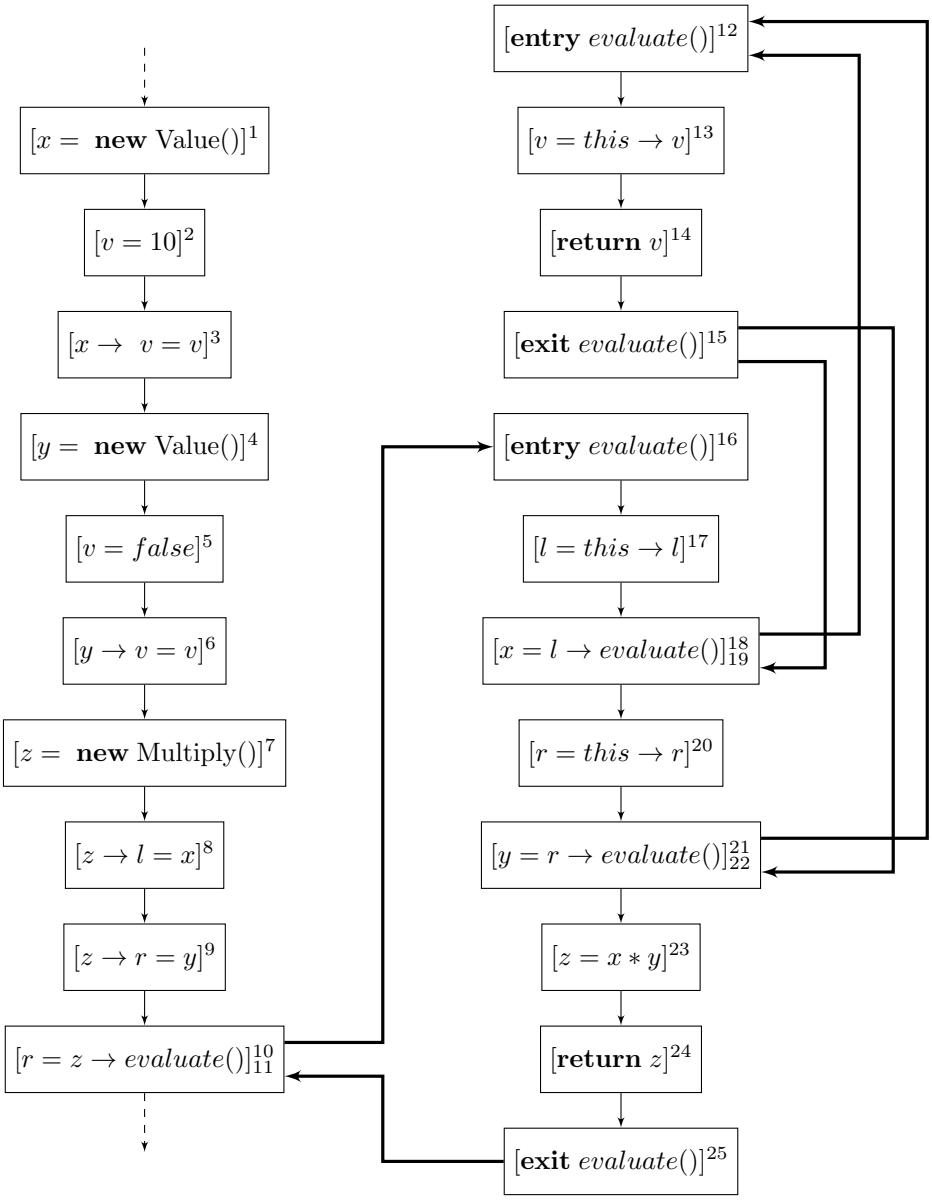

Figure 8: Control flow of the example program in Listing 4.

C Iteration steps of the worklist algorithm

	W	IF	A
1	$[((1,\Lambda),(2,\Lambda))]$	$\emptyset$	$A[1,\Lambda]=$ ([v ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , x ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [])
2	$[((2,\Lambda),(3,\Lambda))]$	$\emptyset$	$A[2,\Lambda]=$ ([v ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [])
3	$[((3,\Lambda),(4,\Lambda))]$	$\emptyset$	$A[3,\Lambda]=$ ([v ↦ (⊥, ⊥, {+}, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [])
4	$[((4,\Lambda),(5,\Lambda))]$	$\emptyset$	$A[4,\Lambda]=$ ([v ↦ (⊥, ⊥, {+}, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥)])
5	$[((5,\Lambda),(6,\Lambda))]$	$\emptyset$	$A[5,\Lambda]=$ ([v ↦ (⊥, ⊥, {+}, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥)])
6	$[((6,\Lambda),(7,\Lambda))]$	$\emptyset$	$A[6,\Lambda]=$ ([v ↦ (⊥, {true}, ⊥, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥)])
7	$[((7,\Lambda),(8,\Lambda))]$	$\emptyset$	$A[7,\Lambda]=$ ([v ↦ (⊥, {true}, ⊥, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥)])
8	$[((8,\Lambda),(9,\Lambda))]$	$\emptyset$	$A[8,\Lambda]=$ ([v ↦ (⊥, {true}, ⊥, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , z ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥]) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥)])
9	$[((9,\Lambda),(10,\Lambda))]$	$\emptyset$	$A[9,\Lambda]=$ ([v ↦ (⊥, {true}, ⊥, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , z ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥]) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥)])
10	$[((10,\Lambda),(16,[7]))$ $,((25,[7]),(11,\Lambda))]$	$\{((10,\Lambda),(16,[7]),(25,[7]),(11,\Lambda))\}$	$A[10,\Lambda]=$([v ↦ (⊥, {true}, ⊥, ⊥, ⊥) , x ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , y ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , z ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])

11	[((16, [7]), (17, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ))}	A[16, [7]]≠ [T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥]) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
12	[((17, [7]), (18, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ))}	A[17, [7]]≠ [T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , l ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , x ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
13	[((18, [7]), (12, [1])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7]))}	A[18, [7]]≠ [T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , x ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
14	[((12, [1]), (13, [1])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7]))}	A[12, [1]]≠ [T ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
15	[((13, [1]), (14, [1])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7]))}	A[13, [1]]≠ [T ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , v ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥]) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
16	[((14, [1]), (15, [1])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7]))}	A[14, [1]]≠ [T ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , v ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
17	[((15, [1]), (19, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7]))}	A[15, [1]]≠ [T ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, {+}, ⊥, ⊥) , v ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
18	[((19, [7]), (20, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7]))}	A[19, [7]]≠ [T ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, {+}, ⊥, ⊥) , v ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])

19	[((20, [7]), (21, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7]))}	$A[20,[7]] \Leftarrow$ [T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥, ⊥) , r ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , x ↦ (⊥, ⊥, {+}, ⊥, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
20	[((21, [7]), (12, [4])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	$A[21,[7]] \Leftarrow$ [T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , r ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , x ↦ (⊥, ⊥, {+}, ⊥, ⊥) , y ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥]) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
21	[((12, [4]), (13, [4])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	$A[12,[4]] \Leftarrow$ [T ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥]) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
22	[((13, [4]), (14, [4])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	$A[13,[4]] \Leftarrow$ [T ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , v ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
23	[((14, [4]), (15, [4])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	$A[14,[4]] \Leftarrow$ [T ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , v ↦ (⊥, {true}, ⊥, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
24	[((15, [4]), (22, [7])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	$A[15,[4]] \Leftarrow$ [T ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, {true}, ⊥, ⊥, ⊥) , v ↦ (⊥, {true}, ⊥, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
25	[((22, [7]), (23, [7])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	$A[22,[7]] \Leftarrow$ [T ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, {true}, ⊥, ⊥, ⊥) , v ↦ (⊥, {true}, ⊥, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
26	[((23, [7]), (24, [7])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	$A[23,[7]] \Leftarrow$ [T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , r ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , x ↦ (⊥, ⊥, {+}, ⊥, ⊥) , y ↦ (⊥, {true}, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, ⊥, ⊤, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])

27	[((24, [7]), (25, [7])) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	A[24, [7]]=([T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, ⊥, ⊤, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , r ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , x ↦ (⊥, ⊥, {+}, ⊥, ⊥) , y ↦ (⊥, {true}, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
28	[((25, [7]), (11, Λ)) , ((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	A[25, [7]]=([T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, {+}, ⊥, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , r ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , x ↦ (⊥, ⊥, {+}, ⊥, ⊥) , y ↦ (⊥, {true}, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
29	[((15, [4]), (22, [7])) , ((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	A[11, Λ]=([T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, {+}, ⊥, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , r ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , x ↦ (⊥, ⊥, {+}, ⊥, ⊥) , y ↦ (⊥, {true}, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
30	[((15, [1]), (19, [7])) , ((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	A[22, [7]]=([T ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, {true}, ⊥, ⊥, ⊥) , v ↦ (⊥, {true}, ⊥, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
31	[((25, [7]), (11, Λ))]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	A[19, [7]]=([T ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, {+}, ⊥, ⊥) , v ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])
32	[]	{ ((10, Λ), (16, [7]), (25, [7]), (11, Λ)) , ((18, [7]), (12, [1]), (15, [1]), (19, [7])) , ((21, [7]), (12, [4]), (15, [4]), (22, [7]))}	A[11, Λ]=([T ↦ ({[7]}, ⊥, ⊥, ⊥, ⊥) , R ↦ (⊥, ⊥, {+}, ⊥, ⊥) , l ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , r ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥) , x ↦ (⊥, ⊥, {+}, ⊥, ⊥) , y ↦ (⊥, {true}, ⊥, ⊥, ⊥) , z ↦ (⊥, ⊥, {+}, ⊥, ⊥)] , [([1],v) ↦ (⊥, ⊥, {+}, ⊥, ⊥) , ([4],v) ↦ (⊥, {true}, ⊥, ⊥, ⊥) , ([7],l) ↦ ({[1]}, ⊥, ⊥, ⊥, ⊥) , ([7],r) ↦ ({[4]}, ⊥, ⊥, ⊥, ⊥)])