Imperial College London
Department of Computing

An Executable Formal Semantics of PHP with Applications to Program Analysis

Daniele Filaretti

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College London and
the Diploma of the Imperial College, 2015

Abstract

Nowadays, many important activities in our lives involve the web. However, the software and protocols on which web applications are based were not designed with the appropriate level of security in mind. Many web applications have reached a level of complexity for which testing, code reviews and human inspection are no longer sufficient quality-assurance guarantees. Tools that employ static analysis techniques are needed in order to explore all possible execution paths through an application and guarantee the absence of undesirable behaviours. To make sure that an analysis captures the properties of interest, and to navigate the trade-offs between efficiency and precision, it is necessary to base the design and the development of static analysis tools on a firm understanding of the language to be analysed. When this underlying knowledge is missing or erroneous, tools can't be trusted no matter what advanced techniques they use to perform their task. In this Thesis, we introduce $\mathbb{KPHP}$, the first executable formal semantics of PHP, one of the most popular languages for server-side web programming. Then, we demonstrate its practical relevance by developing two verification tools, of increasing complexity, on top of it - a simple verifier based on symbolic execution and LTL model checking and a general purpose, fully configurable and extensible static analyser based on Abstract Interpretation. Our LTL-based tool leverages the existing symbolic execution and model checking support offered by $\mathbb{K}$, our semantics framework of choice, and constitutes a first proof-of-concept of the usefulness of our semantics. Our abstract interpreter, on the other hand, represents a more significant and novel contribution to the field of static analysis of dynamic scripting languages (PHP in particular). Although our tool is still a prototype and therefore not well suited for handling large real-world codebases, we demonstrate how our semantics-based, principled approach to the development of verification tools has lead to the design of static analyses that outperform existing tools and approaches, both in terms of supported language features, precision, and breadth of possible applications.

Statement of Originality

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due references are made.

Acknowledgements

I would like to thank my supervisor Sergio Maffeis for offering me this great opportunity and for guiding and supporting me during all the stages of my PhD; my examiners Sophia Drossopoulou and Marco Carbone for their time, feedback and help in improving my work.

I also would like to thank my parents for their patience, understanding and support, and the friends I met here in London for all the fun and the good times.

Finally, I would like to thank EPSRC for their financial support.

'For if one's starting point is something unkonwn, and one's conclusion and intermediate steps are made up of unknowns also, how can the resulting consistency ever by any manner of means become knowldege?'

Plato, The Republic, Book VII

Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Problem description and motivations

1.1 The (in)security of web applications

Many important activities in our lives involve the web. We socialise on Facebook, stay up-to-date on Twitter, have fun on YouTube, bank online, store our work in the cloud, find a job on LinkedIn, find our way home with Google Maps and even attend lectures from world-class instructors from the comfort of hour home. People and companies keep finding new and creative ways of using the web for applications not foreseen by its designers: making phone and video calls, browsing using mobile phones and so on.

The software and protocols on which web applications are based were not designed with the appropriate level of security in mind. In the early days of the Internet, the web consisted of simple web sites that were essentially information repositories containing static documents; web browsers were invented as a means of retrieving this information [1]. The flow of information was one-way, from the server to the browser, and in most cases there was no need for authentication at all. If an attacker compromised the server, she would't typically get access to any restricted information because the data stored on the server was already public.

Today's most used websites are in fact full-blown applications that rely on a two-way flow of sensitive information between the server and the browser. As a matter of fact, modern web

applications, even when developed with high security standards, are likely to contain security vulnerabilities which might cause substantial data or financial loss if discovered and exploited by a malicious attacker [1].

While the importance of security is acknowledged, the most common approach is to enforce security by monitoring the system and intervening when a security violation is detected. As this industry matures, there is a raising awareness that security must be built into the languages and tools used to program web applications, and there is a growing need to gain some level of confidence that an application is effectively secure.

1.2 The role of formal methods

In classic engineering disciplines, products are not released if they contains "bugs". After all, none of us would be happy to buy an automobile along with a list of shortcomings, or walk on a bridge with the warning to be careful because it may collapse. However, softwares are often released along with a list of "known bugs"; far from being surprised when we find software errors, we take them for granted. Whereas with all other products the customer receive a guarantee of reliability, with software we get a disclaimer that the manufacturer is not responsible for any damages due to product errors.

The impact of software errors can vary from unnoticeable (e.g. a wrong pixel on the screen) to severe (e.g. loss of sensitive data) and catastrophic (e.g. failure of a nuclear reactor, aircraft crash, death). For some real-world examples, the reader is referred to [2], a website which documents over 100 "software horror stories". In web applications, software errors may lead to security vulnerabilities which if exploited by a remote attacker may cause data loss, unauthorised accesses, privacy violations and even sabotage of the application itself, with consequent financial losses.

Modern web applications are large (up to millions of lines of code) and complex artefacts built combining different technologies (programming languages, libraries, frameworks etc.) interacting in non trivial ways. Different web applications may well interact with each other on top of different

protocols, standards, encodings, operating systems, etc. Even if each one of these components is claimed to be "secure" or "stable" , subtle (but potentially severe, if discovered and expoloited) glitches are often introduced as a side effect of the interaction between the different components.

It's not surprising that even after extensive testing, code reviews and security assessments conducted by highly qualified individuals, subtle vulnerabilities may remain unnoticed and lurk under the surface for months or even years before being discovered and exploited by an attacker. One of the reasons for this is that testing and related techniques are non exhaustive, i.e. they don't consider all possible execution traces of the application. The more test cases are provided, the greater the likelihood of finding new bugs, but there will always be, in general, a combination of inputs that is missed (considering all possible execution paths is an undecidable problem - it cannot be solved automatically using finite space and memory). When a bug is present in the application but no test input vector triggers it, we have a *false negative*. Non-exhaustive bug finding techniques are able to find bugs, but not to *verify* their absence.

Many web applications have reached a level of complexity for which testing, code reviews and human inspection are no longer sufficient quality-assurance guarantees. Tools that employ static analysis techniques [3, 4, 5] are needed in order explore all possible execution paths through an application and *guarantee* the absence of undesirable behaviours. However, due to classic computability results, this goal can be accomplished only by applying a certain degree of abstraction, with consequent loss of precision. Exhaustive bug-finding tools introduce *false positives*: situations in which the tool reports an error which cannot happen in reality (a false alarm).

Static analysis designers have to face a tradeoff between efficiency and precision. Precise analysers explore all execution paths with very few false alarms, but are in general inefficient and does not scale to real-sized software. On the other, efficient static analysers are also very imprecise and the high number of false alarm issued makes them useless in practice.

Our view is that, in order to ensure that an analysis captures the properties of interest, and to navigate the trade-offs between efficiency and precision, it is necessary to base the design and the development of static analysis tools on a firm understanding of the language to be analysed. When

this underlying knowledge is missing or erroneous, tools can't be trusted no matter what advanced techniques they use to perform their task. In order to correctly reason about programs written in a certain language, it is necessary to first understand *exactly* what each program does, and in order to do that it is necessary, in turn, to understand precisely what each language construct does. Only when the foundations are in place, it becomes possible to use formal methods to systematically construct static analysis or verification tools that are sound by construction, and therefore reliable even in a sensitive context.

1.3 Contributions and thesis outline

In this Thesis, we put this philosophy into practice in the context of the PHP programming language, one of the most common programming languages used by web developers worldwide to build server-side web applications. As discussed above, we aim at starting from the very foundations, by understanding the language in-depth and by encoding our knowledge in a formal way through a mathematical model, known in the field as a *formal semantics*. On top of this knowledge we then build semantics-based verification tools. A first case study is a simple verification tool based on LTL model checking, derived almost "for free" by just leveraging the toolchain offered by $\mathbb{K}$, our semantics framework of choice. The second, more extensive application of our semantics is a general purpose, modular static analysis tool for PHP based on the theory of abstract interpretation, which we systematically applied to our formal semantics in order to derive a static analyser. Both our verification tools are trusted as they are based on the language semantics itself and not on some custom-made assumption about the language behaviour.

The two main contributions of this dissertation are $\mathbb{KPHP}$ and $\mathbb{KPHP}^{\#}$. $\mathbb{KPHP}$ is the first formal (and executable) semantics of PHP to date. The semantics formally encodes our understanding of the language and provides an ideal starting point for the development of a range of tools for the language. The semantics is trusted as we test it against the Zend Test suite, a collection of conformancy tests that comes with the Zend engine, the main (and first) PHP

implementation.

$\mathbb{KPHP}^{\#}$ is a general purpose static analyser based on abstract interpretation, built on top of $\mathbb{KPHP}$. In a sense, $\mathbb{KPHP}^{\#}$ is also a generalisation of $\mathbb{KPHP}$, where the domain of computation has become a parameter and can be freely chosen by the user. In fact, $\mathbb{KPHP}^{\#}$ can be instantiated by providing an *abstract domain* of choice, which produces a corresponding static analyser (e.g. type checking, XSS vulnerability checking, etc.). When the domain of choice is the *concrete domain*, $\mathbb{KPHP}^{\#}$ simply executes programs in the same way that $\mathbb{KPHP}$ does, reproducing the original behaviour. $\mathbb{KPHP}^{\#}$ is general purpose and extensible, in the sense that any user can define her own abstract domain which, when used, will lead to the corresponding analysis. Furthermore, in order to add a new abstract domain it is not necessary to be familiar with the whole formal semantics of the language. In fact, only a small set of operations (such as arithmetics, printing and type conversions) are required to be defined for each abstract domain, while everything else doesn't change.

This dissertation is structured as follows. In the next few chapters we introduce the background and technical material necessary for understanding our contributions. This includes basic information about web applications (chapter 3), an overview of the PHP programming language (chap 3), basic notions of formal semantics (chapter 2) and static analysis (chapter 4) and a tutorial on the $\mathbb{K}$ framework (chapter 2) , the language definitional framework in which we define $\mathbb{KPHP}$. Chapter 5 introduces $\mathbb{KPHP}$, our formal semantics of PHP, while in chapter 6 and 7 we discuss our applications, the verification of PHP based on LTL model checking and our $\mathbb{KPHP}^{\#}$ static analyser respectively. Chapter 8 concludes and discuss future work and possible improvements and extensions to our contributions.

Part II

Background

Chapter 2

Semantics engineering

This chapter introduces the basic ideas and tools used in *semantics engineering*, the practice of developing and maintaining *formal semantics* of programming languages. A formal semantics of a programming language is essentially a mathematical model, which could, in principle, be expressed in many different ways, either on paper or with the aid of automated tools. In this chapter, after introducing the core ideas, we will focus on *mechanised tools* for semantics engineering and we will introduce $\mathbb{K}$, our framework of choice, in great detail. We will conclude the chapter with a detailed survey and comparison of the work that constitutes the current state of the art in the field, with a focus on mechanised semantics (i.e. developed with the aid of some tool) of real-world programming languages.

2.1 Introduction to the formal semantics of programming languages

A (programming) language consists of two components: *syntax* and *semantics*. The syntax specifies the set of sentences that can be written in the language, while the semantics gives them meaning. For example, the string

```
123$bcasknaABC;
```

```
AAAwhile (true ---
```

does not seem to be a valid program in any reasonable programming language, while the string

```
x = "hello world!";
print(x);
```

is indeed a valid program (say, for now, in a generic imperative language $\mathcal{L}$) which, when executed, has the effect of printing the string `"hello world!"`.

In general, given a string of text, deciding wether that string is a valid program in some programming language is, at least *conceptually*, straightforward: the text is either a valid program or not. In the first case it should be correctly parsed and executed while in the second case it should be rejected with an error message. Most programming languages come with a semi-formal description of their syntax, usually as an appendix to the reference manual. When in doubt, one might consult such documentation in order to learn how to write syntactically correct programs. However, when considering the *meaning* of programs, things get more complicated. Let us consider the meaning of the previous example program. One can actually give many different interpretations to it depending on the level of abstraction and the purpose of the observation:

- prints the string "hello world"
- assigns a (string) value to a variable `x`, then prints its content
- passes `x` as argument to the function `print` [1]
- stores a sequence of bytes into a memory area represented by variable `x`. Then, based on this stored values, it produces a series of low-level commands to instruct the graphic card to turn on the pixels forming the text `"hello world!"`.

The example above is obviously harmless, but conveys the idea: the meaning of programs is not always obvious.

[1] In this case we might wonder wether the argument is passed *by value* or *by reference*

For many uses, an informal, approximate english description of a program's behaviour may be enough, but when the domain of application is critical - for example when designing static analysers, compilers, verifiers and other tools - and in general when *reasoning about programs*, we need precise, complete and unambiguous information about the meaning of program constructs. Many existing analysis tools and interpreters are based on an approximate (and sometimes plainly wrong) understanding of the inner functioning of the language they are supposed to execute or support. This is essentially what causes bugs in compilers, incorrect bug finding tools and the like.

To illustrate this, we use an example from [6]. Let us consider the following C function that computes the absolute value of its argument:

```
int abs(int x){
    if (x >= 0) { return x; } else { return -x; }
}
```

The function has been shown to be *correct* by Caduceus [7, 8], an early verification tool for C programs (now part of the Frama-C [9] static analysis and verification framework). However, it contains a bug: calling the function with `MIN_INT` as argument causes an *integer overflow* whose behaviour is *undefined* in C and might cause looping and crashes as well as other issues. This happens because of the underlaying *two's complement* representation of integers. Assuming two-bytes integers, the range of integer values that can be represented in the machine is

$$\mathcal{I}_{16} = [-2^{15}, 2^{15} - 1] = [-32768, 32767]$$

and, therefore, $MIN\_INT = -32768$. However, $-(-32768) = 32768 \notin \mathcal{I}_{16}$, meaning that the number cannot be represented. The verification tool proved a false result (i.e. it claimed the function was correct while it wasn't) because the verification process itself was based on a simplified model of the internal representation of numbers and for this reason wasn't able to detect the erratic behaviour.

According to Winskel [10], *"in giving a formal semantics to a programming language we are*

concerned with building a mathematical model. Its purpose is to serve as a basis for understanding and reasoning about how programs behave".

Formal semantics may be given in different styles and formalisms according to needs, without any particular technique being currently considered better than the others. Historically, the study of the formal semantics of programming languages has followed three main approaches: *operational*, *denotational* and *axiomatic* semantics [10].

Operational semantics, introduced by Plotkin [11], describes the meaning of a programming language by specifying how it executes on an *abstract machine*. For example, once we define program states σ as functions from variables to values, i.e. $\sigma : Var \to Value$, a rule for assignment can be written in operational semantics as follows

$$\frac{(e, \sigma) \to v}{(x := e, \sigma) \to \sigma[x \hookrightarrow v]}$$

where e is an arbitrary expression, x a variable name, v a language value and

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

Intuitively, the rule says that the evaluation of the assignment $x = e$ in a state σ is defined as follow: first, evaluate the right-hand-side expression e to a value v (note that this subexpression is also evaluated in the program state σ); then, update σ so that it maps the variable x to v. Operational semantics are often further categorised into "small-step" and "big-step" according to the *granularity* of the computational steps they capture. For example, the following big-step rule specifies how to compute the sum of two expressions e_1 and e_2 in one single step and immediately produces a value

$$\frac{(e_1, \sigma) \to v_1 \wedge (e_2, \sigma) \to v_2 \wedge v = v_1 + v_2}{(e_1 + e_2, \sigma) \to_e v}$$

The previous assignment example was also written as a big-step rule. However, it is also possible to give rules which capture single steps in the evaluation:

$$\frac{(e_1, \sigma) \to (e_1', \sigma)}{(e_1 + e_2, \sigma) \to (e_1' + e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \to (e_2', \sigma)}{(v + e_2, \sigma) \to (v + e_2', \sigma)}$$

and finally

$$\frac{v = v_1 + v_2}{(v_1 + v_2, \sigma) \to (v, \sigma)}$$

Denotational semantics [10, 12, 13, 14, 15] instead describes the meaning of programs in terms of more mathematical objects chosen from an appropriate *domain*. For example, the meaning of a program P could be described as a partial function $[\![P]\!] : \Sigma \to \Sigma$ mapping program states into program states. Assume the semantics of arithmetic expressions e is given by

$$[\![e]\!] : \Sigma \to Value$$

then the semantics of the assignment statement can be given as

$$[\![x := e]\!] : \{(\sigma, \sigma[x \hookrightarrow v] \mid \sigma \in \Sigma \wedge v = [\![e]\!]\}$$

Finally, axiomatic semantics defines the meaning of program constructs by giving *proof rules* for them [10]. This style of semantics, usually referred to as *program logics*, is best suited to *prove* properties about programs, such as the fact that they are correct with respect to their specification. In this framework, rules are often expressed as *Hoare triples* [16]. As an example, the following Hoare triple for assignment

$$\{B[v/x]\}x := v\{B\}$$

intuitively expresses the fact that, if a logical assertion B holds whenever the variable x is substituted with the value v, then after the assignment $x := v$, B must definitely hold.

As mentioned, no approach is in absolute "better" than another. Instead, different semantic styles may be useful depending on the particular task to be performed. Moreover, these approaches

are related to each other. For example, a program logic (i.e. axiomatic semantics) it typically proven correct with respect to an underlying operational or denotational semantics, while operational and denotational semantics can often be derived from each other [10]. By its very own nature, operational semantics can serve as a guide for implementation while axiomatic semantic is by design well suited for proving program properties: yet, the two semantics can sometimes be proven equivalent [10]. In the remainder of this thesis we only consider operational semantics, which will be introduced in greater detail in the context of the K Framework in section 2.2.2. For an overview of denotational and axiomatic semantics and their interconnection, we refer the reader to [10].

Formal semantics, as originally introduced in the literature, are a theoretical tool, hence *not directly executable*; while useful for program reasoning and language and tool design, they are still just pieces of formal notation written on paper. This means that, once a language or tool is formally specified, its final implementation needs to be realised manually. Moreover, regardless of the style in which they are written, formal semantics of *real-world* programming languages are notoriously large and complex artefacts, difficult to develop and maintain [17, 18, 19, 20]. All these factors contribute to making the implementation phase challenging and error prone, with the result that, in the worst case, "paper semantics" end up being virtually unrelated with the final implementation of the language or tool they specify. As a consequence, even languages or tools equipped with a formal specification are likely to contain bugs and discrepancies w.r.t. their specification.

The approach underlying this thesis, which we have contributed to establish as *best practice* in the field, consists in developing an operational semantics as the same time as the interpreter or tool, and in the same mechanised framework. The interpreter or tool can either be defined and then formally proved correct w.r.t. the operational semantics in a proof assistant like Coq [21] or even automatically obtained from the operational semantics itself, by relying on the execution facilities offered by environments such as Redex [22], Coq [21] and the $\mathbb{K}$ Framework [23]. In the next section we review some of the tools and frameworks available today before introducing $\mathbb{K}$, our

framework of choice, in great detail.

2.2 Tools for semantics engineering

2.2.1 Overview

There is growing awareness of the challenges involved in the design and maintenance of a programming language's formal semantics, and the current trend is all about mechanisation and tool support using proof assistants like Coq [21] and Isabelle/HOL [24] as well as "lightweight" tools such as as PLT Redex [22] and the $\mathbb{K}$ Framework [23]. These systems allow the engineer to define the semantics of a language together with related artefacts such as type systems or analysis tools and offer facilities for execution, debugging, state exploration, theorem proving etc.

Proof assistants provide a logical framework as well as a (typically functional) programming language and allow to state and prove properties about programs written in that language. This feature makes it possible to develop, in a single framework, both the formal specification of a language and its interpreter or tool, and prove the latter correct with respect to the former. In many cases it is even possible to *extract* the program to mainstream languages such as OCaml or Haskell [21, 20], enabling the development of *trusted* interpreters and tools.

Although potentially powerful, this approach has its downsides. Proof assistants are difficult to master and often lack adequate user interface support. Moreover, proofs are large, time consuming and difficult to maintain, not to mention the fact that, in some cases, the idiosyncrasies of the underlying logical framework and its decision procedures might force the semanticist to state her claims in an unnatural/awkward way.

To partly mitigate this, one can use a front-end like Ott [25] which provides a simple BNF-style language for specifying the syntax and semantics of the target language, with the ability of automatically compiling it into theorem prover's code and latex markup. Ott has been used in [26] to formalise a subset of OCaml. This approach is promising, since it allow the semantics engineer to organise everything into a single file so that when the definition is updated, a single

command produces the updated latex markup and theorem prover code. However, we found that using front-ends, while being convenient with small or simple projects, might ultimately introduce further hassle when dealing with large, complex or non-standard definitions (when more ad-hoc hacks are needed).

In order to overcome the difficulties of using full-fledged proof assistants, a category of so-called *lightweight mechanisation tools* [22] emerged. This tools provide lightweight form of mechanisation of the semantics, by performing some form of sanity check of the definitions and providing execution, visualisation and state-space exploration facilities. For example, the tool Redex [22], a domain specific language embedded in Racket [27], produces interactive diagrams showing how terms reduce according to the semantics rules, step by step, and has extensive debugging facilities [22]. The $\mathbb{K}$ Framework is an executable semantics framework based on rewriting logic that provides similar facilities [23] and is increasingly gaining momentum, with languages such as C [6], Java [18], JavaScript [28] and our own PHP [17], among others, being defined within it. We discuss $\mathbb{K}$ in detail in Section 2.2.2.

A limitation of lightweight approaches is that if *meta-proofs* (i.e. proof about the language as a whole) are needed, the designer needs to essentially rebuild the model from scratch in a proof assistant, introducing errors and requiring substantial extra work. On the other hand, lightweight tools allow immediate execution of the semantics, are generally easier to master and provide built-in support for debugging and even some forms of program verification [23, 22]. There is currently a growing interest in the development of *proof assistant backends* for lightweight tools. The latest implementations of the $\mathbb{K}$ framework, for example, offers an experimental Coq backend [18].

2.2.2 The $\mathbb{K}$ Framework

The $\mathbb{K}$ Framework (just $\mathbb{K}$ in the following) is a rewriting-based, executable semantic framework in which programming languages, calculi, as well as type systems and formal analysis tools can be defined [23]. Originally introduced in 2003 as a means to define executable concurrent languages in rewriting logic using Maude [29], $\mathbb{K}$ has increasingly gained popularity among programming

languages researchers as well as enthusiastic language designers. The main features of $\mathbb{K}$ are that it:

- can be used not only to define languages but also related abstractions: type checkers, abstract interpreters, static analysers, etc.
- can define arbitrarily complex features, including the ones found in real languages.
- is modular: adding new features to the language does not require modifying existing (unrelated) features.
- is executable
- offers exhaustive behaviour analysis facilities (e.g. Model Checking).

So far, $\mathbb{K}$ has been used to define formal semantics of real languages (e.g. C [19], Java [18],PHP [17], JavaScript [28], Python [30], Scheme [31], Verilog [32],Haskell [33]) and others as well as several didactic and research languages and analysis tools [34].

The idea behind using rewriting logic as a formalism for the semantics of programming languages is that the execution of a program can be naturally described using rewrite rules. A rewriting theory consists of a signature describing terms and a set of rewrite rules that describe elementary steps of computation. Given some term allowed by signature (e.g., a program together with its input and state infrastructure), deduction consists of the application of the rules to that term. This yields a transition system for any program defined in the language. A single path of rewrites describes the behaviour of an interpreter, while searching all paths (possible via the current $\mathbb{K}$ implementation) yield all possible answers in a nondeterministic program.

However, rewriting logic itself is very general, and doesn't directly tell us how to define programming languages; the $\mathbb{K}$ formalism can be regarded as a convenient front-end to RL designed specifically for defining languages.

An overview of $\mathbb{K}$: IMP

In the remainder of this section we introduce the core concepts of $\mathbb{K}$ by example, considering a simple imperative language called IMP. More detailed information (including step-by-step tutorials and documented semantics of different programming languages and type systems) can be found in [23, 35] or on the $\mathbb{K}$ project website.

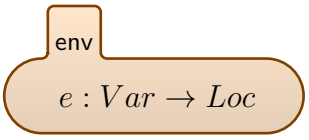
Configurations.

In order to model a programming language in $\mathbb{K}$, the first thing we need to define is a *configuration*. Intuitively, configurations specify the structure of the *abstract machine* on which programs written in the language will be run. For example, a common configuration is one consisting of a program to be executed, an *environment* (i.e. a map from variable names to memory addresses) and a memory (often called *store* or *heap*, which is a map from memory addresses to values). The complexity and structure of configurations entirely depends on both the features of the language and the choices of the designer. Some languages can have extremely large or complex configurations (e.g. the $\mathbb{K}$ configuration for C has about 80 different components [19]), while others, purely based on syntax, like the lambda calculus, can be described by a configuration which consist of the program code only.

$\mathbb{K}$ configurations are represented as labeled, possibly nested multisets, called *cells*. Cells can be defined to contain semantic objects (e.g. lists, sets, maps, etc.) or other cells.

As an example, let us define the configuration for a small imperative language[2]. Essentially we want to instantiate the pattern we discussed before, consisting of a program, an environment and a heap/store. First, we define the environment as a function from variable names to memory locations:

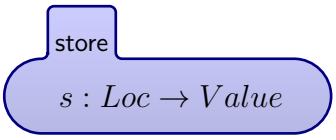
CONFIGURATION:



[2]For now, we consider a simplified language instead of PHP itself, in order to give the reader who is unfamiliar with K a quick and easy understanding of the basic notions. Later on we will move on to PHP.

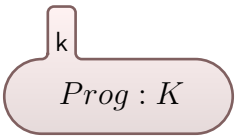
Similarly, the store is defined as a mapping from memory locations to values:

CONFIGURATION:



There is one last thing to be defined: a "container" for the (piece of) program to be run. In $\mathbb{K}$, we accomplish this by putting the program into the K cell:

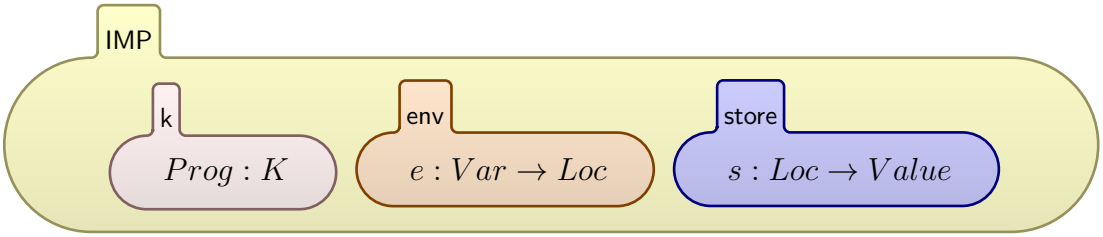
CONFIGURATION:



This deserves some explanation. Cells can contain semantic objects like lists, sets or bags; however, cells can also contain objects of a special sort K of *computations*. For the purpose of this introductory examples, all we need to know is that the program to be executed is stored in a cell called k as a term of sort K. As we'll see shortly, *computations* actually extend the language syntax and introduce a notion of *task sequentiality* via a lists of computational tasks.

We are now ready to give the configuration for our language, IMP, by "wrapping" the three previously defined cells into a top-level container cell (although this is not strictly necessary, it is standard practice in $\mathbb{K}$ as it improves readability and ease of understanding):

CONFIGURATION:



What we have just defined is a general structure, or schema, that concrete (or *ground*) configurations will match. In different words, the configuration we defined above can be though of as a grammar for constructing *configuration terms* for our language. In the $\mathbb{K}$ tool, the same configuration would be written in ASCII as follows:

```
configuration
<IMP>
   <k> $PGM:K </k>
   <env> .Map </env>
   <store> .Map </store>
</IMP>
```

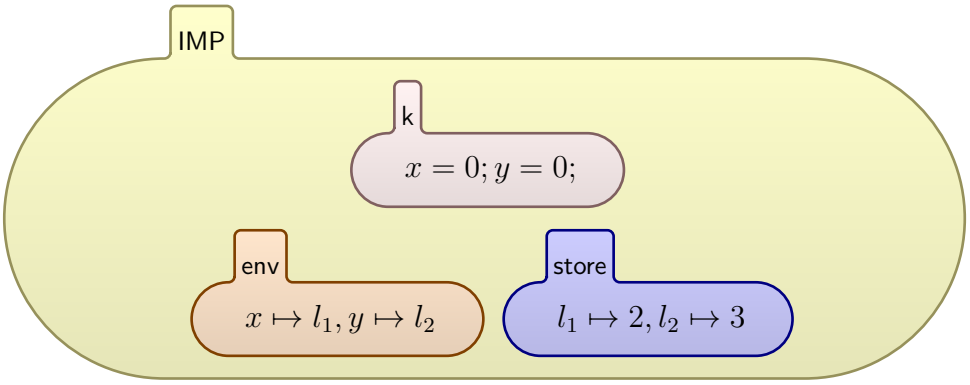
where the special variable PGM will be automatically initialised with the parsed program when the tool is run. Notice also that in the current implementation of $\mathbb{K}$ it is *not* required to explicitly specify the types of maps and other semantic objects (although the correct type can then be enforced in rules).

Assuming a standard syntax for IMP has already been defined and that memory locations are chosen from a countable set $\{l_j\}_{j\in\mathbb{N}}$, the following is an example of concrete configuration, representing a *state* in which the next fragment of program to be executed is

```
x = 0;
y = 0;
```

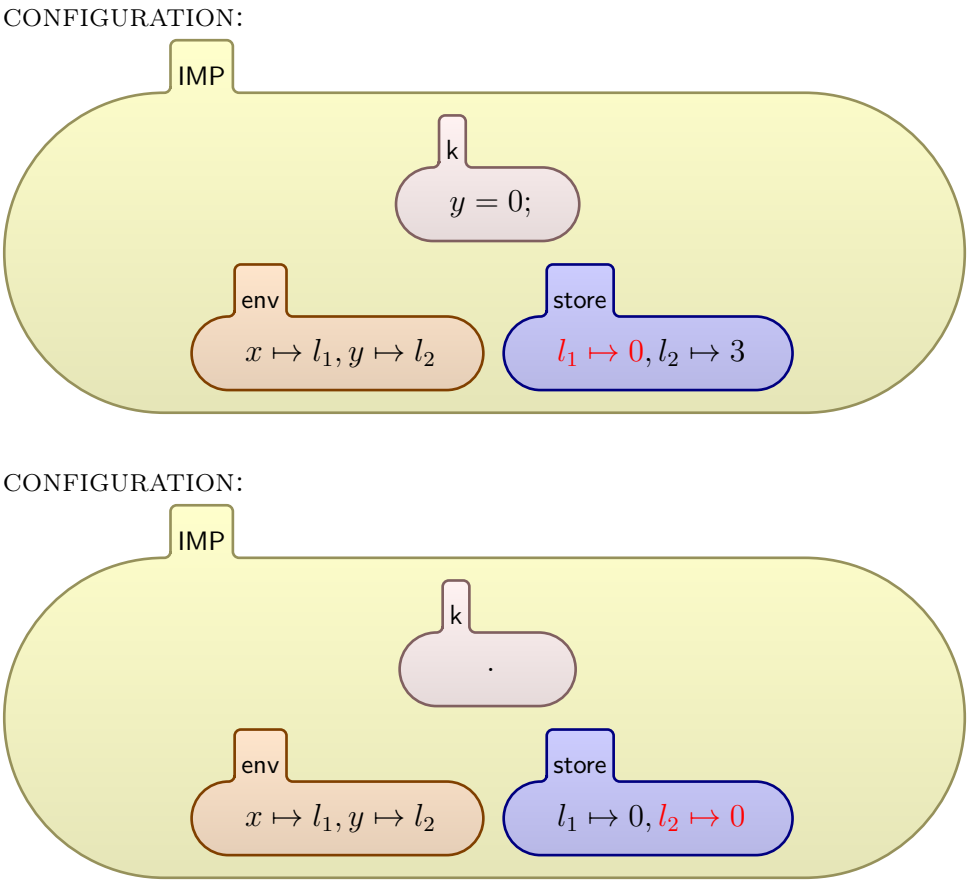
and there are just two variables, x and y, allocated in locations l_1 and l_2 respectively, whose values are 2 and 3:

CONFIGURATION:



Intuitively, we would like that the execution of the program above produces the following sequence of (two) configurations before terminating (we highlight in red the specific parts of the

configuration that changed at each step):



where the symbol "." in the k cell intuitively means that there is nothing left to compute (we will define such notions more formally in the next sections).

Hence, configurations statically represent the state of the abstract machine on which we wish to execute programs. Next, we need a way of specifying how exactly configurations evolve depending on what basic language instruction is to be executed. This is done in $\mathbb{K}$ via $\mathbb{K}$-rewrite rules, or simply $\mathbb{K}$-rules, which are the topic of the next section.

Computations and rules.

The sequencing of evaluation is made possible in $\mathbb{K}$ by *computation structures*. Computation structures, or "computations" for short, extend the abstract syntax of a language with a list structure using the separator $\curvearrowright$ (read "followed by" or "and then"). By convention, $\mathbb{K}$ definitions use a cell named k for holding computations, but nothing prevents us to choose any other name.

$\mathbb{K}$ provides a distinguished sort, K, for computations. The intuition for a computation

$$l_1 \curvearrowright l_2 \curvearrowright \ldots \curvearrowright l_n$$

is that the listed tasks are to be processed in order. The initial computation of an evaluation typically contains the original program as its only task (as seen in the example from the previous section), but rules can then modify it into task sequences. Examples of computations are:

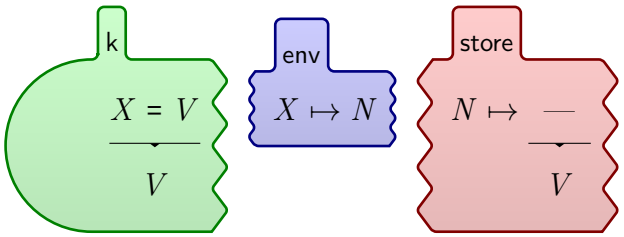
$$x = 0; y = 0; \texttt{while}\ (x \neq 0) x{+}{+};$$
$$x = 0 \curvearrowright y = 0;$$
$$x \curvearrowright \texttt{if}\ (\square \neq 0)\ \texttt{true}$$

All of this will become more clear once we discuss $\mathbb{K}$ rewrite rules, which can be seen as a generalisation of classic rewriting rules [23] and are used to give semantic meaning to languages or calculi in $\mathbb{K}$. We start with a simple example. Recall that IMP configurations consist of a *program* cell, an environment cell and a store cell. Let us consider the elementary operation of assigning a value to a variable (*assignment*):

```
x = "hello world";
```

In IMP, as in most imperative languages, a possible specification of the semantics of this operation is as follows: "*if the next operation to be performed is the assignment of a value V to a variable X, then check if X is actually defined (i.e. associated to a memory cell N in the memory). If so, replace the value stored in the memory address N with V, and return V itself*".

The following $\mathbb{K}$ rule immediately translates this informal description into a very precise one in the context of the configuration we previously defined:



or, in ASCII:

```
rule
   <k> X = V => V ...</k>
   <env> ... X |-> N ... </env>
   <store> ... N => (_ => V) ... </store>
```

Observe how this rule actually resembles a configuration, with some differences: first, although all of the three cells of the original configuration are present, we omitted the external "wrapping" cell. Second, the content of each cell is now composed of two distinct parts: something that is above the line, and something that is below the (optional) line. Third, our cells now have different shapes, sometimes having "zigzag" edges and sometimes not. For comparison, we show the same assignment rule in different formalisms, namely standard rewriting logic:

$$< \texttt{x=v;Pgm} : K >_k < E1 : MapX \mapsto nE1 : Map >_{env} < S1 : MapN \mapsto \texttt{v}S2 : Map >_{store}$$

$$\Longrightarrow$$

$$< \texttt{Pgm} : K >_k < E1 : MapX \mapsto nE1 : Map >_{env} < S1 : MapN \mapsto \texttt{v}S2 : Map >_{store}$$

and structural operational semantics [11]:

$$\frac{env(\texttt{x}) = n \qquad store(n) = \texttt{v}}{env, store, \texttt{x=v} \Downarrow env, store[n \hookleftarrow \texttt{v}], \texttt{v}}$$

However, note how those rules (and in particular the one in standard RL) are generally more verbose that their $\mathbb{K}$-style counterpart and how they require to repeat twice, unchanged, most of

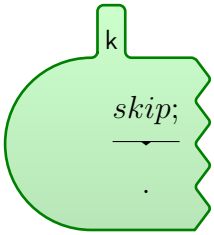
the rule.

In general, for every $\mathbb{K}$ rule we have to define two components: the parts of the configuration we need to *match* and the ones we need to *rewrite* if the match is successful. Everything which is above the line is something that has to be matched, while everything which is below the line, if present, has to be rewritten (replacing what was above the line). The parts of the configuration that are matched without rewriting (like for example the content of the environment cell in the example below) are called *read only* while the other parts are called *read-write*. The parts of the configuration we require to match can also be seen as conditions for the rewriting to apply. Once the match is successful (on *all* the cells mentioned in the rule) the rewrites apply, resulting in a new, updated configuration. The process is then repeated as long as there are rules that apply. When no such rule exists, the computation is *stuck*.

The "zigzag" notation is just a convenient way to express "frames", i.e. parts of the configuration which we don't care about. For example, in the environment cell, we are expressing the fact that we are only interested in finding a mapping from variable name X to a location N (in other words, since we are trying to assign a value to X, we are just interested in wether X is allocated and, in that case, in retrieving its address) [3]. For the program cell the situation is different. In this case, in fact, we don't really want to match on *any* assignment which may be present in the program, but indeed we want the match to be successful if and only if the assignment construct is present at the head of the computation list. In the bubble notation this is expressed by drawing the cell with a rounded left edge (meaning that we are considering the first element of the list). However, the right edge of the program cell is still zigzag, meaning that the rule can (and should!) apply wether the assignment instruction is the last one or not (i.e. we don't care about what instructions are next). If a cell does does not need to be inspected for a transition to happen, it can be omitted. As an example of this, let us consider a `skip` instruction, i.e. an instruction that does nothing. This can be easily defined by the following rule, without mentioning any cell

[3] In case the variable was not declared/initialised, another case of the assignment rule would need to be defined e.g. for allocating the variable on the fly. Alternatively, some languages require the variables to be initialised in advance via an apposite construct

other that the k cell (in fact, the skip instruction can be executed regardless of the content of the memory):



The fact that we can write rules that do not explicitly mention some parts of the configuration is a convenient feature of $\mathbb{K}$ which is called *configuration abstraction*. Essentially, it allows us to specify, for each rule, only the part of the configuration that are needed, ignoring the rest. Although in our example we are really using all of the three cells present in the the configuration, this mechanism turn out to be very powerful when dealing with more complex configurations. For example, in $\mathbb{KPHP}$ (our semantics for the full PHP), the rule for assignment looks very similar to the one we are discussing right now, even though the configuration contains, at the moment, more than forty (40) cells [17]. Other large-scale semantics defined in $\mathbb{K}$ have similar sizes and equally benefit from the feature [18, 19].

Let us now consider a slightly more complex example. Consider the following assignment instruction, where the right-hand-side term has not yet reduced to a value, but it is a complex expression (in this case a given function `foo` – what this function does is irrelevant in this context):

```
x = foo();
```

In this case, the rule we defined earlier for assignment does not apply, since the function call `foo()` is not a value. Instead, we need to evaluate the function (in general, the RHS of the assignment expression) and then, once it has reduced to a value, apply the rule for assignment we defined earlier. In fact, that rule requires the RHS to be a value, but nothing is said about what should be done when this is not the case. What is needed is a set of additional rules specifying the *evaluation* strategy for the assignment construct. In many cases this task can be accomplished easily when using the $\mathbb{K}$ tool's built-in parsing facilities, by adding *strictness* annotations to the syntax production (we describe the $\mathbb{K}$ tool and syntax definitions below). For

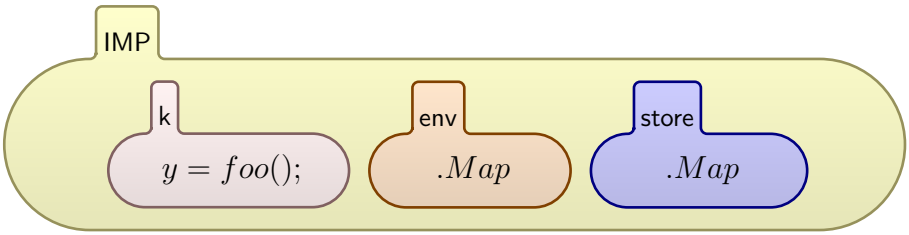
example, the annotation

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

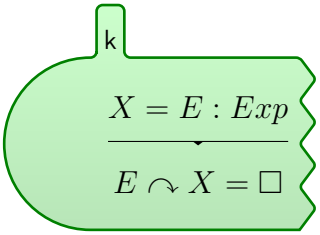
instructs the tool to automatically generate rules which cause the second subterm (the RHS of the assignment) to be evaluated, so that the user will only have to provide the "basic" rule we discussed above. However, the mechanism behind this is worth discussing, because although the automation provided by the tool suffices most of the time, in some cases it might be necessary to manually write such rules in order to express more complex evaluation patterns.

Assume we already have semantic rules available for function call and return, and our abstract machine is in the following initial state

CONFIGURATION:

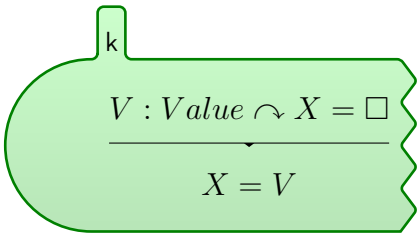


How can such assignment be evaluated? Here is where the notion of *computations*, and *heating/cooling* rules come into place. The first rule to be applied is a *heating* rule of the form



The intuitive meaning of this rule is as follows: "*if the next expression to evaluate is an assignment, and the RHS is a complex expression (i.e. not a value), then evaluate the RHS, and schedule the assignment for later (once the RHS expression has reduced to a value).*" This situation exposes for the first time the fact that, as we mentioned before, the k cell contains a $\curvearrowright$-separated list of tasks.

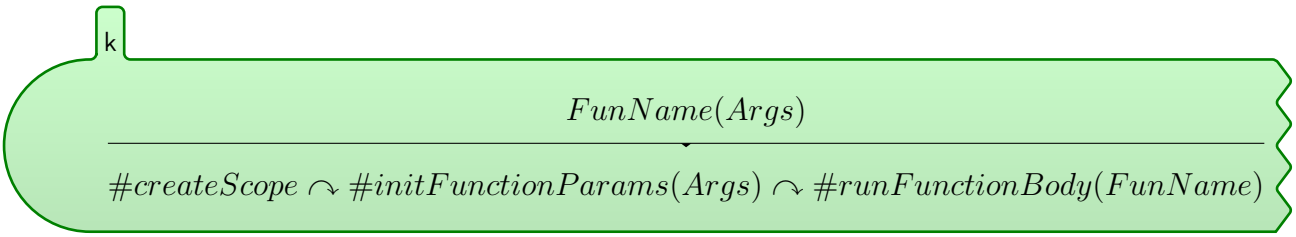
The heating rule actually adds a new task at the top of the list, namely the right-hand-side of the original assignment expression, so that it becomes the next task to be executed. The assignment becomes the second element of the computation list, and the special variable $\square$ is added in place of the original right-hand-side term (the function call). The intuitive meaning of $\square$ is that it is a placeholder for parts of the program which have to be evaluated before being plugged back into their context. The core idea is that once the RHS (expression E in our rule, and the function call foo() in the example) reduces to a value, it is plugged back into the assignment, by writing such a value in place of $\square$. This is what the inverse rule, which is called *cooling rule*, does:



At this point, once the right-hand side of the assignment has been reduced and plugged back into its context, the rule we defined earlier for assignment apply.

This is a very general mechanism in $\mathbb{K}$, and can be used to define simple evaluation strategies (as we have just discussed) as well as more complex ones where, depending on the context, we have different semantics for some operations. In our development we use default heating/cooling rules (as generated by the tool) where appropriate, while we write our own to accomplish more complex or specialised tasks.

While for simple languages all the rules can be defined purely syntactically (i.e. associating for any language construct an atomic operation of the language configuration), in more complex languages this is not always the case. In such situations, $\mathbb{K}$ allows us to define *semantic* tasks, i.e. constructs that are not part of the surface syntax of the language, but can be used for specification purposes in the semantics. We use this mechanisms extensively in our development, but as a preliminary example, let us consider the following (idealised) rule for function call:



Note how we are again explicitly using the computation mechanism, replacing the function call with three sub-tasks, separated by $\curvearrowright$ and hence executed in sequence. This also shows how it is possible to split complex features into simpler ones, and develop programming language definitions in a modular way.

Executing the semantics with the $\mathbb{K}$ Tool. In this Section we briefly show how the notions discussed until now can be put into practice in the $\mathbb{K}$ tool. Note that this discussion is not meant to be a tutorial or user guide to the tool. Since the $\mathbb{K}$ tool is being actively developed (with new releases every few days) and since the project website contains plenty of tutorials and publications, we refer the interested user to such material [35, 23, 34]. Instead, our goal here is just to explain our workflow, as well as illustrate what developing a semantics/tool in $\mathbb{K}$ looks like.

The $\mathbb{K}$ tool, which can be downloaded from `http://k-framework.org/index.php/Main_Page` or run online at `https://fmse.info.uaic.ro/tools/K/`, consists of a set of command line applications, mainly for compiling and executing user definitions, as well as for generating pdf or HTML documentation from the development. We find this last feature quite useful as it allows the developer to maintain the actual formal development synchronised with its latex documentation, which is automatically extracted from the semantics sources.

As mentioned, $\mathbb{K}$ rewriting can be seen as a convenient syntactic sugar over rewriting logic. The workflow is essentially like this: first, a $\mathbb{K}$ definition consisting of a syntax, a configuration, and a set of rules is provided to the tool. The tool, after performing some checks, desugars/compiles such a definition into a standard rewriting logic theory, and produces a parser for the language. At this point, the compiled definition is, in principle, ready to be executed or analysed by any rewriting logic (RL) engine. However, the version of the $\mathbb{K}$ tool in which KPHP is developed uses

Maude as its RL engine.

We now briefly show how a $\mathbb{K}$ definition can be written, compiled and run.

The syntax can be easily defined within the tool using the familiar *BNF* form. The following, for example, is a complete $\mathbb{K}$ syntax definition for IMP:

```
module IMP
  syntax AExp  ::= Int | Id
                 | AExp "/" AExp              [left, strict]
                 > AExp "+" AExp              [left, strict]
                 | "(" AExp ")"               [bracket]

  syntax BExp  ::= Bool
                 | AExp "<=" AExp             [seqstrict]
                 | "!" BExp                   [strict]
                 > BExp "&&" BExp             [left, strict(1)]
                 | "(" BExp ")"               [bracket]

  syntax Block ::= "{" "}"
                 | "{" Stmt "}"

  syntax Stmt  ::= Block
                 | Id "=" AExp ";"            [strict(2)]
                 | "if" "(" BExp ")"
                   Block "else" Block         [strict(1)]
                 | "while" "(" BExp ")" Block
                 > Stmt Stmt                  [left]

  syntax Pgm ::= "int" Ids ";" Stmt
```

```
  syntax Ids ::= List{Id,","}
endmodule
```

Definitions may be enclosed in *modules*. The module system is convenient because it allows us to group individual or related features of the language being developed into several modules which can be imported, tested and debugged independently. As seen, productions may have an attribute, or tag, on the right, enclosed in square brackets. There are many different tags available in the tool, but the most important ones are shown in our examples. The *strict* and *seqstrict* tags are used to specify the desired evaluation strategy for the construct, and causes the tool to generate the appropriate heating/cooling rules under the hood. To further illustrate this, the *strict* tag in

```
AExp "+" AExp              [left, strict]
```

means that the addition operation is strict in both arguments, and we didn't specify an order of evaluation. This means that this operation is actually non-deterministic. In the presence of non determinism, when running a program, the $\mathbb{K}$ tool is able to explore all of the possible answers. Instead, if we want to specify an order of evaluation, we can give an argument to *strict*, for example *strict(1, 2)*. This means that the first argument has to be evaluated *before* the second one. The *seqstrict* tag is just a shortcut for this. The `left` and `right` tags are used to specify associativity, while the `bracket` tag simply tells the tool to treat the given production as a "macro", without generating an AST node for it.

Now that we defined a syntax for our language, we are ready to define a configuration and some rules. As shown, configurations and rules are written in an XML-like notation. We choose to separate configuration and rules from the syntax, by enclosing the formers into the `IMP` module which, in turn, imports the `IMP-SYNTAX` module:

```
module IMP
   imports IMP-SYNTAX
   syntax KResult ::= Int | Bool
```

```
   configuration
   <IMP>
      <k> $PGM:Pgm </k>
      <end> .Map </env>
      <store> .Map </store>
   </IMP>
rule
   <k> X = V => V ...</k>
   <env> ... X |-> N ... </env>
   <store> ... N => (_ => V) ... </store>
// ... more rules here ....
endmodule
```

Once syntax, configuration, and rules have been defined, the definition can be compiled. Assuming that we wrote the definition into a file `language.k`, the compilation is done by executing

```
kompile language.k
```

at the command line. For generating documentation we can type instead

```
kompile --pdf language.k
```

For help about all of the possible options the command is

```
kompile --?
```

The compilation step actually produces a few files, but if the compilation is successful all we have to do in order to run a program using our executable semantics is simply to use `krun`. Assuming we type a program and save it into a file "program.imp", we can simply run it with:

```
krun program.imp
```

`krun` is actually a wrapper that takes many inputs - the program, the parser generated during the compilation phase (or, alternatively, the path to an external parser), and the rewriting logic/maude

definition- and provide a convenient way of running programs using the semantics. If the input program can be parsed, `krun` runs the program according to the semantics, and, if the program terminates, the *configuration* reached at that point is shown. For example, running the program below starting from an empty state (i.e. no variables allocated etc.)

```
int x;
x = 0;
```

may produce a final configuration that looks like:

```
<IMP>
    <k> .K </k>
    <env> x |-> 1 </env>
    <store> 1 |-> 0 </store>
</IMP>
```

While $\mathbb{K}$ supports different notations, in the following we'll mostly use the ASCII for readability and ease of access and comparison to our (and other's) sources.

2.3 Mechanised specifications or real-world programming languages

A real-world language specification may come in many forms: an implementation of the compiler/VM as specification (e.g., PHP); an English definition with varying degrees of rigour (e.g., the C standard [36] and ES5 [37] are fairly precise and complete); a formal mathematical specification (e.g., Standard ML [38]); and a mechanised specification (e.g., the mechanised specification of ML by Lee, Crary, and Harper [39] developed in the Twelf theorem prover [40]).

In this section, we briefly survey the field before performing an in-depth review and comparison of a few selected recent works we believe represent the state-of-the art in mechanised specification

of real-world programming languages. We conclude by discussing where our own work is located in the described scenario and how it contributes to its advancement.

One of the most prominent, fully formalised presentations of a programming language is Standard ML by Milner, Tofte, Harper, and MacQueen [38]. Unlike ML, many real-world languages are designed without formalism in mind. Such languages provide a considerable challenge to mechanisation.

There has been a wide body of work on mechanised language specifications in HOL. For example, Norrish [41] specifies a small-step operational semantics of C in HOL [24], and proves substantial meta-properties of the semantics. Norrish's formalism has not been tested for conformance with implementations. In the CompCert project [42], Blazy and Leroy [43] built a verified optimising compiler for CLight, a significant fragment of C, with a Coq proof that the generated compiled code behaves exactly as prescribed by the semantics of the source program. Several substantial projects build on CompCert: Appel's verified software tool chain [44] combining program verification with verified compilation; Shao's project to certify an OS kernel [45]; Zhao et al.'s verified LLVM [46] which extracts an interpreter from Coq code that is tested using the LLVM regression suite (134 out of 145 runnable tests); and Sewell's CompCertT SO [47], verifying compilation the x86 weak memory model [48].

Inevitably, there is a wide body of work on mechanised specifications of Java and C#: e.g., Syme's HOL semantics [49] of Drossopoulou and Eisenbach's formal Java semantics [50]; the executable formalisation of the C# standard by Börger et al. [51] using Abstract State Machines [52]; and the executable formal semantics of Java 1.4 in rewrite logic by Farzan et al. [53]. We should also mention the formal semantics of Batty et al. on C++ concurrency [54, 55], which is currently having real impact on the C11 standard [36].

For space reasons, we cannot detail all the interesting examples of mechanised specifications of programming languages. In order to focus our review we choose to include only large-scale semantics which are either of *scripting languages*, or are written in $\mathbb{K}$ which is our language definitional framework. Note that we deliberately exclude from our survey all non-mechanised

semantics and semantics that model only a very small subset of the target language.

2.3.1 λ_{JS}

In 2010, Guha *et al.* proposed a novel approach to the formalisation of real-world programming languages based on *desugaring* and applied it to JavaScript [56]. Essentially, they provided λ_{JS}, a small calculus that captures the essential features of JavaScript, together with a *desugaring* function that compiles JavaScript into λ_{JS}. Both λ_{JS} and the desugaring function are defined in the PLT Redex tool which allows them to be debugged and executed.

Although no proof of soundness of the desugaring is provided, the authors validate their system by running a subset of the Mozilla JavaScript Test suite (excluding, as in other work, tests that make use of specialised libraries such as regular expressions etc.). The testing process consisted in executing the same program on λ_{JS} as well as on the major browser engines and observing the difference. We note that, as explained by the authors [56] although the PLT Redex tool automatically provides an interpreter for the semantics, they had to manually write an interpreter for λ_{JS} since the interpreter generated by PLT Redex could not handle the large dimension of some of the test cases.

λ_{JS} has been shown useful as a framework for the construction of safe JavaScript subsets (such as Caja, FBJS etc.). The idea is to first provide a type system for λ_{JS} and to show that typed programs are safe (e.g. they don't attempt to perform an HTTP request). This is straightforward since λ_{JS} is a small language similar to a λ calculus and therefore standard techniques are enough to obtain the results. In order to scale the result to full JavaScript, the idea is to exploit the compositionally of the desugaring function. In order to show that, for example, the JavaScript expression `e1 + e2` is safe, one must show that $desugar(e_1 + e_2) : T$ (i.e. the desugared expression is typed). This can be done by showing that $desugar(e_1) : T$ and $desugar(e_2) : T$ imply $desugar(e_1 + e_2) : T$, which in turn depends on the precise definition of the desugaring rules for the specific operator (`+` in this case). Constructs which admit such a proof can be added to the safe sublanguage.

2.3.2 λ_π

A semantics of a subset of Python, λ_π was defined in [57] following the same methodology introduced by λ_{JS}. Again, the semantics consisted of a core language, λ_π, together with a desugaring function from Python to λ_π. The authors established trust in the development by testing the interpreter (which, as in λ_{JS} was manually written in order overcome the performance issues involved in using PLT Redex's builtin execution engine) against a subset of the official Python conformance test suite. Since λ_π uses the same methodology introduced by λ_{JS} we don't give any more details.

2.3.3 The semantics of C in $\mathbb{K}$

Ellison and Rosu presented an executable formal semantics of C defined in the $\mathbb{K}$ framework [19]. Based on the C99 standard, the semantics models all features of C except for the `_Complex` and `_Imaginary` types and includes only a subset of the standard library. It consists of around 6000 lines of $\mathbb{K}$ source code and 1200 semantic rules. The semantics has been tested against the GCC torture suite [58] and has been reported to pass 99.2% of the test cases [19], more than other compilers such as GCC.

Particular attention has been put in distinguishing between *unspecified*, *undefined* and *implementation defined* behaviour. Unspecified behaviour is for example the order of execution of the operands of a binary operators such as `a + b`, which is non deterministic. An example of implementation defined behaviour is the size of `int`. Finally, an example of undefined behaviour is the dereferencing of a pointer to a non allocated memory area. A further example of undefined behaviour in C was discussed in section 2.1. The semantics is parametric w.r.t. implementation defined behaviour, allows non-determinism to deal with unspecified behaviour and put great effort in *not* modelling undefined behaviour.

This approach, in contrasts with real-world C compilers that happily execute undefined programs, makes sure that programs containing undefined behaviour are given no meaning by the semantics and therefore get stuck (with appropriate debug information) at runtime. This technique can be used to check wether a given C program is "*conforming*" according to the C99

standard, meaning it doesn't exhibit undefined behaviour. Being based on $\mathbb{K}$/Maude, the semantics also provide *state-space exploration* and *explicit state LTL model checking* facilities which are used in order to explore non-determinism and check simple liveness and safety properties of C programs.

2.3.4 JSCert

Bodin *et al.* introduced JSCert [20], a mechanised specification of the ECMA5 standard in the Coq proof assistant together with JSRef, a reference interpreter for JavaScript defined in Coq and extracted to OCaml. The semantics is defined as a Coq inductive definition in a *pretty-big-step* style [59]. Pretty-big-step semantics was recently introduced by Charguéraud as a variation of big-step that mitigates the rule duplication issues that arise when considering non-termination and exceptional behaviour. In order to establish trust in their semantics, the authors give a Coq proof of the correctness of JSRef w.r.t. JSCert. Moreover, they extensively test the reference interpreter against `test262`, the ECMA conformance test suite.

JSRef passes 1796 out of 2782 tests associated with the core language (chapters 8-14 of the ECMA5 standard). Failures are either due to the use of `for-in` (which is not covered by their semantics), bugs in the parser (which is a third-party parser) or calls to libraries that are not supported by their system. Where the ECMA standard left implementation details unspecified, they made arbitrary but hopefully natural choices. The task of parametrising the semantics in order to model different browser behaviours is left for future work.

Possible applications (which haven't been realised yet) include investigating the properties of JavaScript fragments used for secure sandboxing and developing certified compilers and analysis tools. As a side-effect of the development of JSCert/JSRef (and the associated effort to fully understand JavaScript), the authors discovered bugs in all major browser implementations, in the ECMA5 and ECMA6 standards and in the test suite itself.

During the initial phase of his doctoral studies, the author of this dissertation has personally contributed to the JSCert project by developing an earlier version of a safety proof for the language

(in Coq) as well as several semantics rules. The experience maturated while working on the project has influenced the author's personal view on the topics of semantics engineering and related tools, and eventually influenced him in moving towards a lightweight mechanisation tool like $\mathbb{K}$. Clearly, the advantage of JSCert over the related work surveyed in this section (including our own) is the ability to perform language-level proofs and extract certified OCaml code. On the other hand, using this tool chain is not straightforward and has a steep learning curve. In this regard we note that the JSCert team consisted of eight people (3 of them were Coq expert) and it took around one year of work to obtain the results. Large-scale formalisations of real-world programming languages developed in lightweight tools require less time and effort, and provide many features for free (such as execution) allowing the engineers to focus more on the language issues instead as on developing interpreters and proofs or dealing with complex formalisms.

2.3.5 $\mathbb{K}$ Java

Bogdanas *et al.*recently developed $\mathbb{K}$-Java [18], an executable formal semantics for Java 1.4 written in the $\mathbb{K}$ framework. The semantics is based on the *Java Language Specification* [60] and covers all of the language features. As usual, libraries are left out of the equation - only a few of them are modelled on a per-need basis. Mimicking the JLS, the semantics is split into two parts: a static and a dynamic semantics. The static semantics perform computation that could be done statically according to the JSL and returns a preprocessed AST which is passed to the dynamic semantics for execution.

Like other $\mathbb{K}$ semantics, $\mathbb{K}$-Java is directly executable and was thoroughly tested against the behaviour of the Oracle JDK in order to asses its correctness. The authors also developed a conformance test suite of more that 800 tests aimed at exercising every language feature including corner cases and unusual combinations. The choice to develop their own test suite was initially motivated by the fact that for Java, there is currently no *publicly available* conformance test suite. Java's official test suite, the Oracle Java Compatibility Kit (JCK), is only made available by Oracle under certain conditions, and the authors were not able to obtain it [18].

As other $\mathbb{K}$ semantics, $\mathbb{K}$ Java benefits from the ability to perform state-space exploration, symbolic execution and explicit-state LTL model checking. The authors show how to use this toolchain to verify properties of (multi-threaded) Java programs. Moreover, the authors see $\mathbb{K}$ Java as a possible reference implementation as well as a platform for experimentation with language extension.

2.3.6 $\mathbb{K}$ JS

Another formal semantics for JavaScript, $\mathbb{K}$-JS has been recently proposed by Park *et al.*in [28]. Being defined in $\mathbb{K}$, the semantics is directly executable and, as in [20], it is tested against the core language section of the `test262` test suite.

$\mathbb{K}$ JS successfully overcome some of the limitations of previous JavaScript semantics. First, it passes 100% of the test suite, while the other semantics only passes a subset of it. Moreover, it manages to successfully define "problem" constricts such as `for-in`. The problem with `for-in` (and the reason why it was not defined in [20] is that it is not completely defined in the ECMA standard, which leaves the iteration order unspecified, allowing for non-determinism. $\mathbb{K}$ JS instead defines the iteration in `for-in` in a non-deterministic way, using the set-theoretical `choose` operation for choosing the next element to be taken. Using the state-space exploration facilities offered by the $\mathbb{K}$ tool and its support for non-determinism, the semantics of a program can then be explored to reveal all of the possible outcomes of a `for-in` loop. In an attempt to make the semantics even more readable and more closely related to the ECMA specification, the authors define a meta-language for the semantics which closely resembles the language used in the specification. They then develop their semantics by using the meta-language, achieving an almost $1:1$ correspondence with the pseudo-code contained in the ECMA specification.

As other $\mathbb{K}$ definitions, $\mathbb{K}$ JS takes advantage of state-space exploration and symbolic execution as well as recent advances in the $\mathbb{K}$ framework which make it possible to perform Hoare-style deductive verification via reachability-logic [61]. The authors demonstrate how to find previously known vulnerabilities using symbolic execution and how to prove functional correctness of data-

structure manipulating programs via deductive verification. Furthermore, the semantics is designed to get stuck and report an error when behaviour that is underspecified in ECMA5 is encountered. This behaviour is normally hard to detect since each browser essentially make different implementation choices regarding those behaviours. For this reason, $\mathbb{K}$-JS could be trivially used to detect non-portable code (i.e. code containing underspecified behaviour) by simply running a program in it. If the program gets stuck, then the behaviour is underspecified in the standard. During the development of $\mathbb{K}$ JS the authors found a number of behaviours specified in ECMA5 which are not tested by any of the tests contained in `test262`, for which they wrote their own handwritten tests.

2.3.7 Conclusion

Recent work has shown that, with appropriate tools and methodologies, the mechanised formalisation of real-world programming languages, once considered a daunting task, is now possible.

However, while the mechanised formalisations of languages such as C and Java can already be considered "mature", with new work providing incremental improvement over the previous, the situation is different for scripting languages such as JavaScript and PHP, which present additional challenges due to their highly dynamic nature.

While JavaScript has been already formalised in the past, we are the first to provide a formalisation of PHP in [17]. We find this surprising, since there exists a rich literature on the static analysis of PHP (such as for finding security vulnerabilities), and we find odd that nobody realised that with a formal semantics many existing approaches and techniques could be improved. The only previous work we are aware of that looks in depth as some aspect of PHP semantics is an analysis of the array-copying mechanism of PHP by Tozawa *et al.* [62]. They formalise a tiny fragment of the language that suffices to describe the array copy mechanism, and show a flaw in a runtime optimisation used by the Zend engine. Their work sheds light on how complex the array semantics in PHP is, and was an inspiration for us to dig deeper into the PHP semantics.

Our formalisation of PHP in $\mathbb{K}$ has been the second large-scale formal semantics of a real-world programming language to be defined in $\mathbb{K}$ (after the semantics of C by Rosu *et al.*), influencing

	Framework	Executable	LOC	Based on ref. test suite	Based on lang. spec	Applications
C in $\mathbb{K}$	$\mathbb{K}$	yes	6000	yes	yes	Detect undefinedness. Model checking. non-determinism exploration
JSCert	Coq	no	5000 (3000 JSCert 2000 JSRef)	yes	yes	-
λ_{JS}	Redex	yes*	1250 (400 semantics 800 desugaring)	yes*	no	design of safe JS subsets.
$\mathbb{K}$ Java	$\mathbb{K}$	yes	6000	N/A	yes	model checking multi-threaded programs
$\mathbb{KPHP}$	$\mathbb{K}$	yes	6000	yes	N/A	Symbolic execution. Model checking of security properties abstract interpretation
λ_{π}	Redex	yes*	2000 (redex) 10000 implement.	yes*	N/A	-
$\mathbb{K}$ JS	$\mathbb{K}$	yes	7000 + libraries	yes	yes	detect non-portable JS code non-determinism exploration symbolic execution Deductive verification

Table 2.1: A comparison of recent mechanised specification of real-world programming languages. While the PLT Redex tool is capable of executing mechanised semantics (similar to $\mathbb{K}$), in both λ_{JS} and λ_{π}, the authors replaced the generated interpreter with a handmade one,for performance. In the same works, they tested their development against the reference test suite (respectively of JavaScript and Python) but on a selection of hand-picked cases. In all other cases, the authors attempted a more systematical testing.

subsequent work such as $\mathbb{K}$ Java [18] and $\mathbb{K}$ JS [28]. Furthermore, our work pioneered the use of the $\mathbb{K}$ framework for the formalisation of dynamic scripting languages.

The majority of the work we reviewed is based on the official language specification as well as the official test suite provided by the language. PHP (and Python) constitute a notable exception to this pattern. Since no official specification was available at the time our work was developed, we had to deal with additional challenges such as the modelling of the memory model essentially via trial-and-error. This, together with the (almost) complete lack of previous work, influenced the total development time of $\mathbb{KPHP}$, which we observed to be higher than that of other $\mathbb{K}$ semantics. We observed that mechanised specifications built using lightweight mechanisation tools are easier to develop and maintain (when compared to specifications built in proof assistants such as [20]), are executable, and provide debugging, state-space exploration and verification tools for free. Those "helper" features, combined with more user-friendly development environments and efficient workflows, ultimately make it possible to develop large-scale formalisations of real-world programming languages in realistic time and effort.

Chapter 3

Web Applications

Web applications employ an array of different technologies to implement their functionalities. This chapter is a short introduction to the most important ones. It does not mean to be exhaustive; we will only discuss the topics we believe are necessary for the understanding of this dissertation. For more in depth and up-to-date information we will provide the interested reader with additional references.

3.1 Basics

3.1.1 The HTTP protocol

HyperText Transfer Protocol [63] is the communication protocol used to access the World Wide Web. We might not be aware of it, but when interacting with a web application, our web browser is actually exchanging HTTP messages with it. Originally developed for retrieving static text-based information, the protocol has been extended during the years in order to accommodate new features, but its structure didn't change. HTTP uses a message-based model in which a *client* sends a *request* message to a *server* which, in turn, replies with a *response*. HTTP messages consist of two parts: a *header*, containing control information, and a *body*, containing the message itself. Figure 3.2 shows an example in which the user types the address of a banking website and

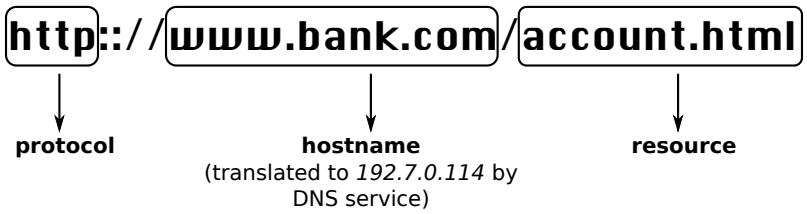


Figure 3.1: An URL

receives a login page. To start the transaction, the user types (or clicks on) a *Universal Resource Locator* (URL) such as

```
http://www.bank.com/account.html
```

into a web browser. The URL is composed of three parts, as shown in figure 3.1: the protocol to be used, the hostname and the path to a requested resource. The first part, `http`, tells the browser that the protocol to be used is the HTTP protocol, while the hostname `www.bank.com` is passed to a Domain Name Service (DNS) which returns a 32 bit unique identifier for the server, its *IP address* [64]. The third part specifies the server resource to be requested. With this information, the browser assembles an HTTP request and sends it to the server via its IP address. The physical delivery of the data over the network is performed by the underlying Transmission Control Protocol (TCP) [65]. The generated request might look like the following

```
GET /account.html HTTP/1.0
Host: www.bank.com
```

where the server associated to `www.bank.com` is asked to retrieve the resource `/account.html`. `GET` is only one of the possible *methods* available, but it is, together with `POST`, one of the most widely used. `HTTP/1.1` communicates the version of the protocol being used, 1.1 in this case. Since the `GET` method's function is essentially to retrieve information, the request's body is empty. The server replies with the following response

```
HTTP/1.0 200 OK
<HTML> ... </HTML>
```

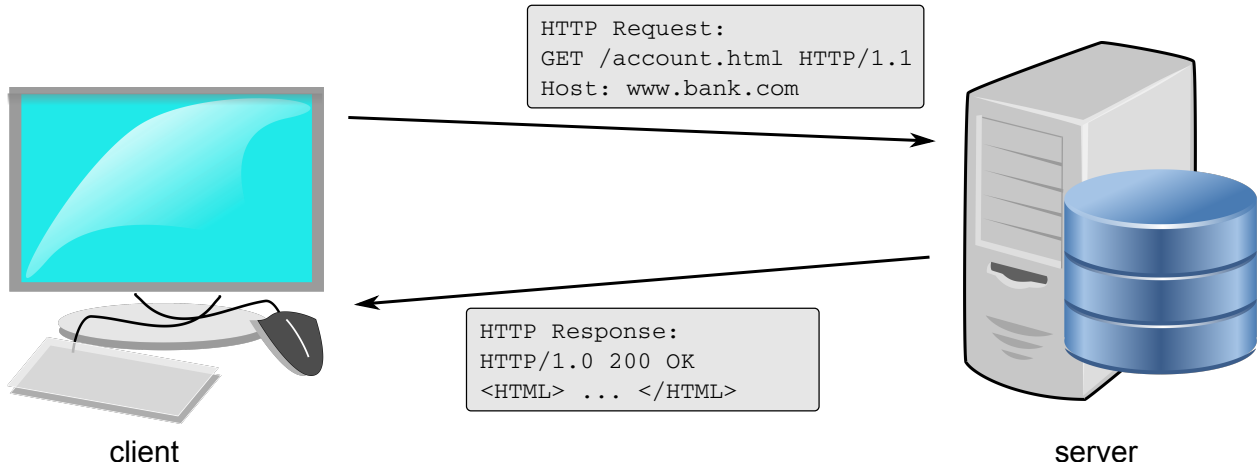


Figure 3.2: Communication via HTTP protocol

consisting of a confirmation code (`HTTP/1.0 200 OK`) and the requested resource - the bank's HTML page. Finally, the client's browser renders the HTML with the effect of displaying the page on the user's screen. The resource requested by the `GET` method can also be used to pass parameters to the server as in the following example

```
GET /login.php?usr="dnf" & pswd="123" HTTP/1.1
Host: www.bank.com
```

where the parameters named `usr` and `pswd` are passed to the resource `/login.html` with values `"dnf"` and `"123"` respectively. From the user's perspective, this can be achieved by appending the parameters in the URL:

```
http:://www.bank.com/login.php?usr="dnf" & pswd="123"
```

Notice that, like any other URL, this could be bookmarked or shared between users. However, if the data stored in the parameters is sensitive, this might lead to privacy violation attacks such as history sniffing [66].

Another commonly used HTTP method is `POST`, which differs from `GET` in the fact that the parameters are passed in the message body instead of being encoded in the URL. While `GET`'s functionality is to retrieve resources, `POST` is commonly used to *perform actions* [1]. A common scenario is that of a web page containing a form:

```
<html>
<head> <title>My Page</title> </head>
<body>
        <form action="http://www.bank.com/check-login.php" method="POST">
                User: <input type="text" value="usr">
                Password: <input type="text" value="pswd">
                <input type="submit" value="send">
        </form>
</body>
</html>
```

When the user sends the form by clicking the "send" button, a `POST` request containing the form data in the body is assembled by the browser and sent to a resource which validates and processes the form:

```
POST /check-login.php HTTP/1.1
Host: www.bank.com
{usr : "dnf", pswd : "123"}
```

HTTP provides many other methods and parameters which we do not discuss here as not strictly relevant for the understanding of the following. For more details, we refer the interested reader to [1] and [67].

3.1.2 Client and server-side technologies

In the previous section we discussed how the HTTP protocol allows communication on the web and how parameters can be passed to a remote resource via the `GET` and `POST` methods. The complex functionalities and rich user experience offered by modern web applications are possible because of a vast spectrum of technologies deployed on both clients and servers. For example, in order to display a page in a browser, we need an implementation of HTML, CSS, and JavaScript, among others. On the server side, we might (and actually do!) want to respond to a request by generating content *on-the-fly*, tailored to the individual user depending on the input parameters,

as opposed to simply retrieving static content as in the early web. Indeed, this is precisely the reason why it is useful to pass parameters to the server: if all resources were static, there would be no need for parameters. For example, after a user submits her login data, the server responds with a different page depending on wether the login attempt was successful or not. The transaction typically starts with an HTTP POST request to a *dynamic resource* on the server, as shown in the previous example, where the request's body contains user name and password. However, this resource will typically not be a static web page but a program written in some language (e.g. PHP, Java, ASP, etc.). The program will retrieve the data, establish a connection with a database server (which might be a remote server itself), perform an SQL query incorporating the parameters sent in the request, check the result and finally output either a welcome or error page, depending on the results of the query.

3.1.3 State: cookies and sessions

HTTP is a *stateless* protocol, i.e. it treats each request as an independent transaction that is unrelated to any previous request, meaning that the communication consists of independent pairs of requests and responses. However, in order to implement most modern functionalities, web applications need to track the state of each user's interaction across multiple requests. A typical example is an online shop where users can browse products and add them to a shopping basket. After a user adds one or more products, she might reload the page or simply open another section of the shop (e.g. a different product category) while expecting the basket to still contain her products. To enable this kind of functionality, the application must somehow maintain a set of data generated by the user's actions across several requests. There are two approaches to this: *cookies* and *sessions*.

The cookie mechanism enables the server to send items of data to the client, which the client stores and resubmits to the server at every subsequent request unless specific action is taken (e.g. the cookie is deleted from the client or modified by the server). To issue a cookie the server uses the `Set-Cookie` response header:

```
Set-Cookie: price=8.99
```

The user's browser then automatically adds the following header to subsequent requests back to the *same* server:

```
Cookie: price=8.99
```

In addition to the cookie's actual value, the `Set-Cookie` header usually includes optional attributes such as `expires`, `domain`, `path`, `secure` and `HttpOnly`. Those attributes control how the browser handles the cookie and are frequently reported as security sensitive (i.e. they are manipulated in order to perform attacks). However, we'll not discuss them here, as their treatment is out of our scope. The interested reader is referred to [1], [68] and [67].

Sessions were introduced to solve the main problem with cookies: that data is stored on the client, and hence completely under control of the (potentially malicious) user. With sessions, the web application itself keeps track of the data necessary to perform the required functionality (e.g. the shopping cart). This data is stored within a server-side structure called a *session*. When a user performs an action, such as adding an item to the shopping cart, the web application updates the relevant details within the user's session. When the user later views the contents of her cart, data from the session is used to return the correct information. In order to associate each client with its session, a unique identifier, the *session ID* is issued by the server and sent to the client as a cookie. The client, in turn, will send the session ID back to the server at every subsequent request, allowing the server to identify the user and provide the correct state information. The reader should note that, while all information is now stored on the server, an attacker might still be interested in stealing another user's session ID in order to impersonate that user (as discussed in 3.3.2).

3.2 PHP

PHP was originally created in 1994 by Rasmus Lerdorf as a simple set of Common Gateway Interface (CGI) binaries written in the C programming language [69]. Originally used for simple

tasks such as tracking visits to his online resume, he named the suite of scripts "Personal Home Page Tools," or "PHP Tools" for short. Over time, the suite was expanded and more functionality such as database integration was added, allowing users to develop simple forums or guestbooks. In 1995 Lerdorf released the sources of PHP Tools to the public. In 1997, Andi Gutmans and Zeev Suraski approached Rasmus online and discussed various aspects of the current implementation of PHP. In an effort to improve the engine, Andi, Rasmus and Zeev decided to collaborate in the development of a new programming language. This entirely new language was renamed simply 'PHP', with the meaning becoming a recursive acronym - PHP: Hypertext Preprocessor. The language runtime became known as the *Zend Engine* [70] and, for many years and through its various versions, it has been the only mainstream implementation of PHP. PHP, now in version 5, has grown to a fully-fledged dynamic scripting language featuring object-oriented facilities, complex array and object iterations, automatic type conversions, aliasing and more. Recently, alternative implementations appeared [71, 72, 73]. Some of them, and notably Facebook's HHVM [72] are starting to enjoy some popularity thanks to websites such as Wikipedia starting to make the switch (mainly for performance reasons) [74] from Zend. In 2014, Facebook released the first draft of an English-language specification for the language [75] (briefly discussed at the end of this section and referenced as necessary in chapter 5). However, the Zend Engine is still the most popular PHP runtime as well as the reference point on which all alternative implementations and specifications are based. Accordingly, all of our discussions and results, unless stated otherwise, will be based on the Zend Engine.

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

All the examples are reproducible by pasting the code in the PHP Zend Interpreter (version 5.3.26 or similar) available in most OS X or Linux distributions. The symbol > precedes the shell output.

3.2.1 Hello World Wide Web

PHP scripts are typically run by web servers. As discussed in Chapter 3, typing a URL such as

```
http://example.com/hello.php?name=xyz
```

in a browser may cause an HTTP request such as

```
GET hello.php?name=xyz HTTP/1.1
Host: example.com
```

to be sent, which in turns may cause the responding server to invoke PHP on the file `hello.php` listed below:

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

This minimal example illustrates the typical behaviour of a PHP script. It receives inputs from the web and it responds by generating an HTML page depending on such inputs. The predefined `$_GET` array is in fact populated from the parameters of the HTTP request, `name` in this case,and `echo` is a simple output command that generates the body of the HTTP response message. The client's browser will render this page and visualise the string

```
Hello xyz
```

Incidentally, the `hello.php` script above also constitutes a minimal example of PHP code vulnerable to Cross Site Scripting, a common security vulnerability affecting numerous web applications (introduced and discussed in detail in Section 3.3.2). Suppose an attacker crafts an URL such as

```
http://example.com/hello.php?name=<script> evil() </script>
```

and tricks a user to visit it. As a result, the following webpage will be rendered in the user's browser

```
<HTML>
   <Body>
      <script>
         evil();
      </script>
   </Body>
</HTML>
```

Since the page contain a script, the browser will execute it with the effect of calling the `evil()` function and triggering the attacker's malicious payload.

In case of HTTP requests using POST, the predefined array `$_POST` would be used instead. Analogously, the arrays `$_COOKIES` and `$_SESSION` are populated with cookies and session information respectively. Builtin PHP functions such as `start_session`, `cookie`, `close_session`, allow session manipulation without directly manipulating HTTP responses.

In this thesis, we focus on PHP as a programming language, and leave the interesting topic of formalisation of the server execution model to future work.

3.2.2 A Closer Look

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

Aliasing and references

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

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

Aliasing can be useful for example to write functions that operate on parameters containing large data structures, avoiding the overhead of copying the data structure inside the local scope of the function. On the other hand, aliasing is notoriously difficult to analyse statically [76, 77]. PHP references are different from pointers (as in C) in that neither address-arithmetic nor access to arbitrary memory is allowed. For example, the following code would be rejected:

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

Braced and variable variables

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

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

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

Variable variables are useful for example to simulate higher-order behaviour by passing functions *by name*. On the other hand, they hinder static analyses, because it is not possible in general to determine statically the set of variables used by a PHP script.

A similar argument applies to braced variables, a syntax to turn the result of an arbitrary expression into an identifier, as for example in

```
${"x"} = "y";            // equivalent to $x = "y"
$z -> {"x".$x};          // equivalent to $z -> xy
```

Type juggling

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

integers `1` and `0`.

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

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

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

Arrays

Arrays in PHP are essentially *ordered* maps from `integer` or `string` *keys* to language values. If a value of a different type is given as a key, type juggling will try to convert it to an `integer`. `$x = array("foo"=> "bar",4.5 => "baz");` The array `$x` above maps `"foo"` to `"bar"` and `4` to `"baz"`. Note how the `float` value `4.5` was automatically converted to the `integer` value `4`. In general, array keys are converted to either `int` or `string` according to a set of non-trivial *type juggling* rules. For example, the code

```
$x = array( 1=>"foo", "2"=>"wow", 3.5=>"doh", "4.5"=>"omg",  NULL=>"lol" );
var_dump($x);
```

produces the following output:

```
array(5) {
  [1]=>
  string(3) "foo"
  [2]=>
  string(3) "wow"
  [3]=>
  string(3) "doh"
  ["4.5"]=>
```

```
  string(3) "omg"
  [""]=>
  string(3) "lol"
}
```

Array elements can be accessed via standard square-bracket notation, and it is also possible to assign an element to an array without specifying a key.

```
$x[] = "default"        // use default key 4
echo $x[4]              // prints "default"
```

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

```
echo current($x);       // prints "foo"
next($x);               // advances the pointer
echo current($x);       // prints "wow"
```

Objects

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

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

Access to an array element is always granted, whereas access to an object property is regulated by the visibility attribute, and depends on the context whence the property is being accessed. Inheritance is class-based. Consider the classes `par` and `chld` defined in figure 3.3 (the example is adapted from the Zend test suite [78]). Crucially, the code

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

```
class par {
        private $id = "foo";
        function displayMe() {
                echo $this -> id; }}

class chld extends par {
        public $id = "bar";
        public function displayHim() {
                parent::displayMe(); }}
```

Figure 3.3: Two simple classes

returns `"foo"` because `$id` is declared `private` in the superclass `par`.

If instead `par` defined `$id` as `public`, the code would return `"bar"`. In Section 5.2.1, we shall see how we capture this subtlety in our semantics by indexing the arrays of object fields by *key-visibility* pairs. A notable difference between objects and arrays is that objects are copied by reference whereas arrays are copied by value. Most existing analysis tools for PHP do not support objects, because their semantics is not easy to analyze.

Global variables as array properties

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

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

Because of this ambivalence of global variables, in the semantics it is natural to model scopes as heap-allocated arrays, rather than as frames of a stack independent from the heap. This is analogous to what happens in JavaScript semantics [79, 20], where global variables are the properties of the global object, and scopes are heap-allocated objects. Maffeis *et al.* [80, 81] show

that confusing variables (which can usually be identified statically) with object properties (which can be computed at run-time) complicates security analyses for JavaScript. The case for PHP is even more desperate, as "thanks" to variable variables even variables on their own cannot be determined statically.

3.2.3 Facebook's PHP specification

In July 2014, Facebook released the first draft of their PHP specification [75], one day before we presented our work on $\mathbb{KPHP}$ [17] at ECOOP 2014. The specification, available online and open source, is written in informal English and is meant to provide a set of guidelines to perspective PHP implementors, where we believe substantial effort has been spent in trying to make the specification as much general as possible (to give more freedom to implementors) while also setting a few boundaries to ensure a fair share of compliancy and agreement between implementations. We now mention a few fact about the specification, and how it compares with our work.

The memory model, being ultimately inspired by the Zend implementation, is similar to ours, although more abstract. While we tried to mimic exactly the memory model as found in Zend, Facebook's specification gives more freedom to the implementors by explaining that "*A conforming implementation may use whatever approach is desired as long as from any testable viewpoint it appears to behave as if it follows this abstract model*" [75]. One important difference between our memory model and theirs is that, while we store every value inside a `Zval` (as described in the official PHP documentation [82] and discussed in chapter 5), they describe different "value containers" such as `VStore` for scalar variables and `HStore` for arrays and objects. However our model and theirs also share some similarities such as the fact that objects are represented in a very similar way to arrays. Overall, our detailed memory model is simply an instance of theirs, which is left more generic and abstract on purpose.

We found, in general, the specification to be very informal, not only compared to our formal semantics (which would be expected) but also to other informal specifications such the ECMA specification for JavaScript [37]. Furthermore, most of the challenging features we struggled to

(successfully) model in our semantics, such as array copy, complex evaluation order and corner cases of `foreach`, are left out of the equation and instead regarded as "implementation dependent". Similarly to us, they do not model the copy-on-write mechanism (discussed in 5.2.3), which is also considered as an implementation choice. They however mention that, if implemented, copy-on-write must be unobservable from the user point of view and provide a set of guidelines for "compliant" copy-on-write strategies. In a blog post called *Critiquing Facebook's new PHP spec* [83], Paul Biggar identified 18 points in the specification which are left out as implementation-dependent. While we share the view of keeping the good parts of the language while trying do discourage the use of the bad ones, we also interpret this choice as a way to justify the fact that their own HHVM [72] sometimes behave differently from Zend. Throughout chapter 5, when describing our semantics, we will provide references or comparisons with the specification when appropriate and applicable.

3.3 Web security

Like any new technology, web applications brought a new range of security threats with them. From the early days, the set of most commonly found vulnerabilities changed over time; some kinds of flaws, such as SQL injection, slowly become less prevalent as awareness increased, while others such as Cross Site Request Forgery (CSRF) seem to be increasing [84]. Instead, vulnerabilities such as Cross Site Scripting seem to be as prevalent now as they were a few years ago [84]. Throughout this evolution, compromises of prominent web applications have regularly hit the news and there is currently no indication of a decreasing trend [1, 67]

3.3.1 Overview

The purpose and impact of an attack varies depending on the functionality of the application itself, the nature of the data it manages, its security and other factors. In a typical worst-case scenario, an attacker gains unrestricted access to an application's backend and steals sensitive data

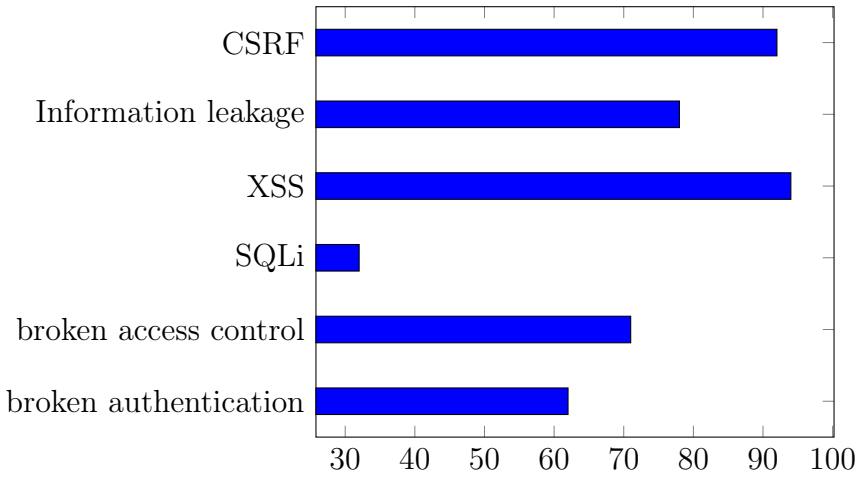


Figure 3.4: Percentage of web applications containing different classes of security vulnerabilities.

such as bank accounts, medical records, or perhaps industry or government secrets. In other cases the purpose of the attack might be to sabotage the application causing some down-time. This attack is known as *denial of service (DOS)* [85] and can have have large financial consequences for companies whose profit depends on their online presence and 24/7 availability. Other kinds of attacks can be more subtle. For example an attacker could abuse a web application so that she is able to purchase a product at a lower price, or orchestrate an attack whose goal is to steal another user's credentials (such as the session ID for some service she is currently logged in). While those classes of attacks do not compromise the web application in its entirety, they can still undermine a company's reputation (customers don't like to transact with a company whose website is vulnerable and poses them under threat) and cause financial loss.

In [1], the authors performed penetration testing on a large number of web applications and found that most of them were actually vulnerable, in one way or another. Figure 3.4 shows what percentage of web applications tested during 2007 and 2011 were found to be affected by some common categories of vulnerability:

Broken authentication comprises defects within the application's login functionality. When exploited, it might enable an attacker to guess passwords, launch brute force attacks, or bypass the login.

Broken access control involves scenarios where the application doesn't properly protect access to its data and functionality. This might potentially enable an attacker to view other users' data or perform privileged actions.

SQL injection enables an attacker's malicious input to interfere with the application's interaction with back-end databases. An attacker may be able to retrieve arbitrary data from the application, or execute commands on the database server itself (e.g. deleting or compromising data)

Cross Site Scripting (XSS) enables an attacker to target other users of the application, potentially gaining access to their data and performing unauthorized actions on their behalf.

Information leakage involves cases where an application leaks information that an attacker might use in developing an assault against the application.

Cross-site request forgery allows a malicious web site visited by the victim to interact with the application to perform actions on behalf of the user that the user did not intend.

An official list of the ten most prevalent web application vulnerabilities is provided yearly by OWASP, the Open Web Application Security Project in [86]. A summary of the latest data is shown in Table 3.1.

Ultimately, a fundamental source of insecurity for web applications is that, since the client is outside of the application's control, users can submit arbitrary and potentially malicious data to the server-side application. At a very high level, attacking an application consists in crafting such data so that it interferes with the application's normal behaviour allowing attackers to perform malicious or unauthorised actions. For this reason, web applications must consider all user data as potentially malicious, and are responsible for protecting themselves against possible attacks. In other words, web applications not only have to provide the desired functionality, but also need to take steps to ensure attackers cannot submit malicious inputs to interfere with their expected logic and behaviour.

	Owasp Top 10 - 2013
A1	Injection
A2	Broken Authentication and session management
A3	Cross Site Scripting (XSS)
A4	Insecure Direct Object References
A5	Security Misconfiguration
A6	Sensitive Data Exposure
A7	Missing Function Level Access Control
A8	Cross Site Request Forgery (CSRF)
A9	Using Known Vulnerable Components
A10	Unvalidated Redirects and Forwards

Table 3.1: The OWASP top 10, reproduced from `https://www.owasp.org/index.php/Top_10_2013-Top_10`

3.3.2 Taint style vulnerabilities

Among the many classes of web application vulnerabilities, which we cannot discuss in full detail and generality here, we are mostly interested in the class of *taint-style* vulnerabilities. Tainted data originates from potentially malicious users and can cause security issues at vulnerable points in the program, known as *sensitive sinks* [76]. Tainted data enter the program at specific positions (usually, where user input is gathered) and spread via assignments and similar instructions. If tainted data reaches potentially dangerous positions (such as SQL queries or system calls) without being previously *untainted* (or *sanitised)*, then there exists a taint-style vulnerability. In the program, tainted data can be sanitized using a set of suitable operations to remove the dangerous properties. Many important vulnerabilities such as cross-site scripting, SQL injection and script injection can be seen as instances of this general class [76]. Each class of taint-style vulnerabilities can therefore be described by a set of *entry points*, *sanitisation routines* and *sensitive sinks*.

Cross Site Scripting (XSS)

Cross Site Scripting is by some measure the most prevalent web application vulnerability found in the wild [76, 1, 86, 67]. Despite being well known and understood, it is still found in the majority of web applications, including security-critical ones such as those used by online banks. Its purpose

is to *attack other users* of a vulnerable web application, usually by stealing their credentials, spreading a worm or by making them unable to access the website itself.

In order to introduce XSS let us first consider a typical *session theft* scenario where an attacker manages to steal a legitimate user's authentication cookie for a particular web application and therefore is able to impersonate the user for that session. Suppose Joe is currently logged to his bank's website. Bob wants to steal his session ID in order to impersonate Joe and steal some of his money. Bob could set up a malicious web page that, when loaded, executes a JavaScript program that sends the user's cookies to his server. Then, he could trick Joe to open the page, for example by sending a link to him. This kind of attack is motivated by the fact that each request containing an authentication cookie is treated by the server as a request of the corresponding user, as long as the same user didn't log out.

However, this attack scenario is difficult to implement because of two factors. First, web browsers implement a mechanism, known as the *same origin policy* [87], that makes it difficult to steal cookies in the first place. The same origin policy has been introduced after it has been realised that while web browsers automatically send a cookie only to the web site that issued it it, with JavaScript it is possible and easy to send cookies to arbitrary locations (e.g. the attacker's server). The same origin policy restricts the access rights of JavaScript programs by ensuring that each program can only access the cookies that were issued to the web page that loaded the script.

Second, most web applications now implement some form of *CSRF-protection*, which, even assuming the attacker managed to steal another user's session, makes it difficult for her to perform unauthorised actions on behalf of the legitimate user. One such measure consists in paying attention to the HTTP *referer header* [63, 88]. When Bob issues the following request to `www.bank.com`

```
GET /transfer.php?usr=amount="5000"; to="evil-bob" HTTP/1.1
Host: www.bank.com
Referer: www.attacker.com
```

the bank will reject the transaction because the referer header tells it that the request originated from `www.attacker.com`. Basically, the bank could adopt a policy that ensures that all requests

are originating from `www.bank.com` itself and possibly a restricted set of other trusted origins. Any request coming from a different origin will be rejected. However, we note that the referer header is optional and actually not used by many browsers and that for this reasons many web applications do indeed accept *all* requests missing the referrer header. Moreover, an attacker could relatively easily manipulate the referer header or even eliminate it in order to take advantage of the above mentioned mechanism. Another option consists in requiring more information to be sent as part of the request. This might include solving a captha, requesting additional information from the user (such as her date-of-birth or possibly a PIN or password) or even considering only incoming HTTP requests containing a particular secret token [1, 67] which is passed, for example, as a hidden form field and is not known to the attacker. This secret information could be embedded in every link or form sent from `www.bank.com` to the user. The bank could then accept only those requests that includes the secret token.

XSS attacks circumvent both the same-origin policy and anti-CSRF mechanisms by injecting malicious JavaScript into the output of trusted but vulnerable applications. When this happens, the malicious code appears to originate from the trusted site and for this reason has complete access to all (sensitive) data related to this site.

Suppose Joe's banking website has a search functionality provided by the following script:

```
<?php
        echo "You␣searched␣for␣" . $_GET['search'];
?>
```

where the user's search query is supplied to the script as a `GET` parameter called `search`. All Bob has to do to steal Joe's credential is to trick him to click on a URL such as the following (e.g. by sending it to him via email):

```
http://vulnerable.com/post.php?s=<script>document.location =
    'evilserver.com/steal.php'+document.cookie</script>
```

which causes Joe's cookies to be sent to Bob's server at `evilserver.com`. The same origin policy would not protect Joe here, because the malicious code originates from the trusted banking ap-

plication itself[1]. Similarly, the malicious code could initiate a bank transaction on behalf of Joe as seen before, but this time the referer header would indicate that the request originated from `www.bank.com` itself, and for this reason will not be blocked.

The particular type of XSS vulnerability discussed above is called *reflected XSS* [1, 67], since the attacker's malicious input is immediately returned (i.e., reflected) to the victim. Another variant of XSS, called *stored XSS* [1, 67], occurs where the application first stores the malicious input on the server (in a database or the file system) and than, at a later stage, returns it to users upon their request. A typical example is that of a web forum or guestbook where an attacker leaves a comment containing malicious JavaScript. Any user who will read that comment in the future will also execute the code. A famous example of stored XSS attack is the MySpace/Samy worm [89] that appeared in 2005 on the MySpace [90] social networking site. The author of the worm managed to embed a JavaScript program into his MySpace profile by cleverly bypassing MySpace's filters. Users who visited his profile would execute the code which essentially made them friend with him, displayed the message "after all, Samy is my hero" and finally copied the code on their profile, so that every user visiting that profile would get infected as well. Samy went from 73 to over one million friends in less than one day [89].

In general, an XSS vulnerability is present in a web application when user content (JavaScript in the previous example) received by the application is not properly handled (e.g. removed or sanitised) before being incorporated into the output sent back to a user. Taking the viewpoint of a server running a PHP web application, XSS can be characterised as a taint-style vulnerability as follows:

- *Entry points* into the program: `GET`, `POST` and `COOKIE` arrays.
- *Sanitization routines*: language-dependent functions designed to remove harmful properties from data. In PHP, for example they are functions such as `htmlentities()` and `htmlspecialchars()` which escape possibly harmful characters such as `<` and `>` from the argument in order to prevent the browser from interpreting them as HTML/JavaScript when

[1] In practice the attack string may need to be URL encoded.

commands. However, also type casts can be used as sanitizers. For example, casting a string to integer immediately turns that data into non dangerous data.

- *Sensitive sinks*: Routines that display data on the screen, e.g. `echo()`, `print()` and `printf()`.

SQL injection (SQLi)

The main purpose of an SQLi attack is to gain unauthorised access to a web application's backend database. Consequences range from theft of sensitive data (financial, medical, governative) to financial losses and sabotage (e.g. by erasing a company or government's database).

The interaction with the database is commonly performed by means of SQL queries [1, 9, 67]. SQL queries are often assembled dynamically by combining predefined SQL constructs together with user input. The following example query retrieves an account based on a name and password that are provided by the user through an `HTTP GET` request:

```
mysql_query("SELECT * FROM users WHERE name='$_GET[name]' AND pw='$_GET[pw]'");
```

An SQL injection vulnerability occurs whenever an attacker is able to alter the syntactic structure of an SQL query in an unexpected way [76, 91, 1, 67]. In the above example, by providing the string

```
admin' OR 1 == 1; --
```

to the `name` parameter together with any value for the password field (which is irrelevant for this example), an attacker could trick the application into issuing the following query to the database:

```
SELECT * FROM users WHERE name='admin' OR 1 == 1; -- AND pw=whocares
```

However, the sequence `"--"` denotes the start of a comment in SQL, with the result that everything that follows it will be ignored. The effect of the resulting query is to return the account of the user 'admin' (or any other user for that matter), bypassing the application's authentication mechanism. Similarly, the attacker could completely destroy all user data by supplying the following value for `name`:

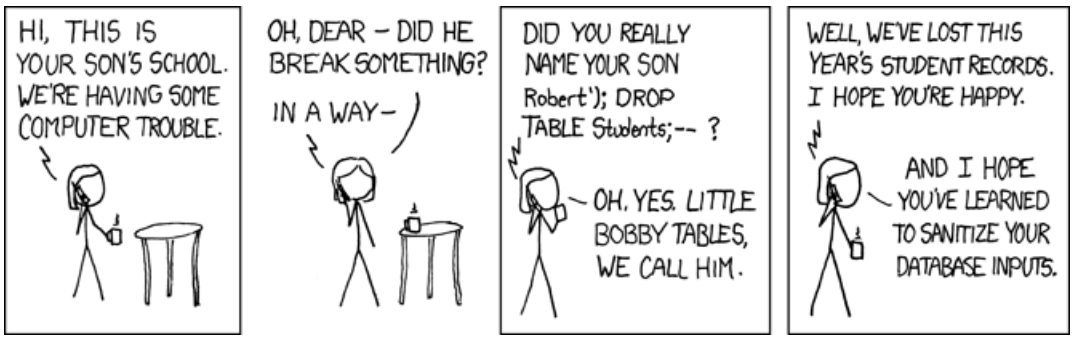


Figure 3.5: A popular comic strip about SQL injection.

```
frank' OR 1 == 1; DROP TABLE users; --
```

Although different in purpose from XSS, SQLi also belongs to the class of taint-style vulnerabilities and can be characterised as follows:

- *Entry points* into the program: `GET`, `POST` and `COOKIE` arrays.
- *Sanitization routines*: functions designed to escape parts of the input that could interfere with the intended meaning of a SQL query. One example in PHP is the library function `mysql_escape_string`, which has similar purpose to `htmlentities()` and `htmlspecialchars()` as seen for XSS.
- *Sensitive sinks*: Routines that use data to perform SQL queries such as `mysql_query()`

Figure 3.5 shows a popular comic strip depicting a realistic SQLi attack scenario [92].

Chapter 4

Static analysis and security

Current mainstream practices for software assurance mainly consist in *testing* and *code auditing*. Testing consists in running a program on a set of inputs, ensuring that it behaves as expected. The more successful test cases, the higher the confidence in the fact that the program will be *correct* (according to its intended behaviour). If a test fails, the developer normally looks for the offending bug and tries to fix it before re-running the test suite. Testing is convenient in the sense that when a test fails, it indicates a real issue in the program - there are no *false alarms*. On the other hand, testing is costly, time consuming and, worst, *non-exhaustive*: in general, no matter how many tests are executed for a given program, testing itself will fail to explore all the possible execution paths that a program may take at runtime.

Code auditing essentially consists in convincing someone else, usually a fellow developer, that some code is correct. An advantage over testing is that when reading code, humans can generalise and infer facts, which help them reasoning about multiple executions at once. Similar to testing, code auditing is costly and non-exhaustive. Both testing and code auditing are unable to explore all of the possible execution paths a program may take at runtime and therefore they might miss errors.

Being able to consider as many execution paths as possible is important for security. One single failure, if found and exploited by a skilled attacker, may be enough to cause serious damage.

Attackers dedicate time and effort to discover and exploit hard-to-find faults that weren't caught by the application's developers themselves. In the light of this, it makes sense to investigate techniques that allow many or all execution paths to be explored at once.

Static analysis consists in analysing a program without running it. In this, it resembles code auditing but with the fundamental difference that instead of being performed by a human, the code review is performed by a program, the *static analyser*. This has an important advantage: static analysis is able to provide higher coverage than testing and code auditing, and in some cases it can even explore *all* possible execution paths in a program and *guarantee* the absence of errors. This benefits come at a price: static analysers can only check for limited (and generally simple) properties; they may miss errors *or* issue false alarms; furthermore, their use may be time consuming.

In this chapter we introduce the basics of static analysis and we survey existing technologies for the static analysis of web applications. In Section 4.1 we explain what static analysis is and why it is challenging. In Section 4.2 we discuss the important concepts of *soundness* and *completeness* as well as other design tradeoffs such as precision versus scalability. In Section 4.3 we explore the idea of approximating the set of program behaviours and we discuss the previously introduced concept under this point of view. in section 4.4 we introduce basic concepts such as control flow graphs, basic static analysis algorithms as well as flow, path and context sensitivity and see semi-formally how static analysis work in practice. In Section 4.5 we formally introduce *abstract interpretation*, a theory of approximation of sets and functions over sets which has been successfully used to understand and design not only static analysers but also other formal methods such as model checking and deductive verification. Finally, in Section 4.7 we survey how static analysis has been used in the context of web application security, focusing on tools targeting PHP.

4.1 The nature of static analysis

A static analysis is an algorithm which takes a program as input and attempts to provide an answer to a question about the set of its possible runtime behaviours. Typical questions include:

- is every user input sanitised before it flows into a sensitive sink?
- is every SQL query in the program constructed using *only trusted* input?
- is every array index in bound?
- is every pointer which is dereferenced non-null?
- is it true that divisions-by-zero cannot occur?

Unfortunately, answering those questions *exactly*, automatically and for any program is *undecidable*. To explain this, we take a step back and consider a question which might be considered the "canonical" static analysis problem: the *halting problem*. Let $\mathcal{X}$ be an arbitrary program. Does $\mathcal{X}$ terminate on all inputs? Can we design an algorithm that answer the question for any program? Unfortunately the answer is negative, because the existence of such an algorithm leads to logical contradictions and is therefore impossible.

To give an intuition of this, let us assume such an algorithm, say `termination`, exists. Given a program `prog`, `termination(prog)` returns exactly `true` if `prog` terminates on all possible execution environments (or inputs) and `false` otherwise. Let us then consider the following program `P`

```
P =  while (termination(P)) do
                skip;
```

Does `P` always terminate? There are two cases to consider:

- assume `P` terminates. Then, our procedure `termination(P)` must return `true`. But, for the definition of `P`, this causes `P` to enter an infinite loop and therefore `P` does not terminate.
- On the other hand, let's assume `P` does not terminate. Then, `termination(P)` must return `false`. But, for how `P` is defined, this causes `P` to terminate since the `while` loop is exited immediately.

Since in both cases we reached a contradiction, our assumption about the existence of `termination` (which is the only external assumption we introduced in our reasoning) must have been wrong, which proves that a program such as `termination` cannot exists.

It is also possible to show that most of the interesting static analysis questions (including the examples above), are equivalent to an instance of the *halting problem* and therefore must be themselves undecidable. This is usually done using a *proof by transformation/reduction*. Assume for example the existence of an analyser that precisely detects all out-of-bound array accesses in a given program. Now let's take an arbitrary program and apply the following transformations to it:

- replace every array access such as `$a[$i]` with a conditional expression such as for example `$i >= 0 & $i < $a.len() ? $x[$i]; exit()`
- replace every exit point in the program with an out-of-bound access such as `$a[$a.len() + 10]`

If the error-checker finds an out-of-bounds error, then it has determined that the original program halts. Otherwise, it has determined that the original program does not halt. This reasoning shows that if analyser can find out-of-bound errors than it can solve the halting problem and vice versa. For this reason we must conclude that guaranteeing the safety of array bounds and deciding program termination are, somehow, "equivalent in difficulty".

The example shows that it is impossible to design an analyser that answers a question about the program for all possible inputs in an *exact* way - if that were possible, the analyser would be solving the halting problem, which we already know to be impossible. This, however, doesn't mean that static analysis is impossible. While exact static analysis is impossible, *useful* although approximated static analysis is indeed possible, and an active field of research. Approximation manifests itself as a combination of the following:

- the analyser might not terminate
- the analyser might raise false alarms (false positives)

- the analyser might miss errors and claim the program to be "safe" while it's not (false negatives)

We analyse this tradeoffs in more detail in the next section.

4.2 Design tradeoffs

When designing an analysis, it is possible to influence how approximation manifests itself in the analysis result by making the appropriate design choices and tradeoffs. For instance, non-terminating analyses are generally avoided for practical reasons. In this cases, termination is usually enforced by introducing a loss of information. The balancing of false positives and false negatives leads us to the discussion of the concepts of *soundness* and *completeness*. We also introduce an important tradeoff in static analysis: precision versus scalability.

4.2.1 Soundness vs completeness

A sound (or correct) static analysis proves true facts about the program semantics (intended as the set of all possible runtime behaviours) of the program it is given as input. For example, if a sound analyser says that there are no out-of-bound array access in the program being analysed, then the user can be sure that the program will indeed *not* exhibit out-of-bound accesses at runtime. In general, if a sound analysis says property P is true about a program semantics, then P is actually true. However, according to this definition, the fact that P is true does *not* necessarily imply that the analysis claims that P is true. For example, a program might be free from division-by-zero errors (in the sense that there is no execution path that leads to a division where the denominator is zero) but the analysis might fail to discover that and instead raise a false alarm.

On the other end of the spectrum are *complete* analyses. A complete analysis is such that, if property P about the semantics of the program is true, then the analysis claims that P is true. For example, if the program is free from index-out-of-bound errors, then the analysis will be able to detect it, and therefore there are *no* false alarms. On the other hand, nothing is said about the

opposite: if the analysis says property P is true, there is no guarantee that P will be true. If the analysis says a program is free from a certain class of bugs, then it might not be the case. In other words, a complete analysis might miss errors.

In summary, a sound analysis is able to verify the absence of unsafe behaviour but is not able to find bugs with certainty: if a sound analysis finds a bug, it might be a false alarm, but if it claims the program is error free then it must be error free. Complete analyses, on the other hand, cannot prove the program to be error free, but can detect bugs without false alarms: if a complete analysis claims the program is safe, then it might not be true, but if it finds a bug, then that bug must be a real bug. The situation is depicted in figure 4.1. A sound *and* complete analysis is able

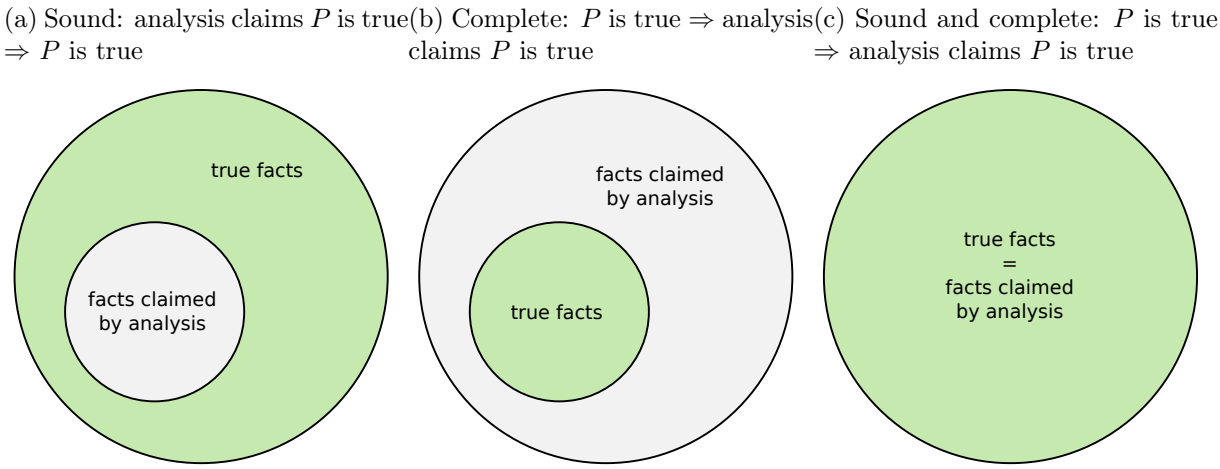


Figure 4.1: Sound, complete and sound *and* complete static analyses

to verify the absence of errors without producing false alarms. A property P about a program holds if and only if the analysis claims the fact. However, realising a sound and complete analyser would amount to solving the halting problem and therefore is impossible. *Every* static analysis must lie somewhere in the space between soundness and completeness. In practice, sound analyses are used when the primary goal is guaranteeing the absence of bugs (for example as in critical embedded applications), while complete analyses are used when the priority is on finding real bugs (such as in non-critical application development). Many interesting analyses try to reach the sweet spot of finding many bugs with little false positives by being nor sound nor complete,

4.2.2 Precision vs scalability

Another design tradeoff in static analysis is between *precision* and *scalability*. A precise analysis models program behaviour very closely in order to reduce the number of false alarms (in case of a sound analysis) or to reduce the number of missed bugs (in case of a complete analysis). A scalable analysis is able to analyse large codebases within reasonable time and space resources. Unfortunately, very precise analyses also have a larger algorithmic complexity which makes them less efficient and ultimately less scalable. On the other hand, a very efficient analysis will also be less precise with the result of producing a large number of false positives (negatives). Again, achieving both goals is impossible. Let us consider a sound static analysis. If we increase the precision of the analysis we reduce the number of false positives. However, we also get progressively closer to a sound analysis which does not produce any false positive, which is undecidable. Similarly, if we progressively increase the precision of a complete analysis we'll end up catching exactly all real bugs and therefore we'll have obtained an undecidable analysis.

Balancing this tradeoffs is often considered more as an art than as a science, and the perfect combination depends also on the particular language being analysed, its usage and the average size of programs in the chosen application domain.

4.3 Approximating all possible runtime behaviours

Ultimately, all static analysers attempt to statically compute (a machine representation of) the set of possible runtime behaviours of the input program. This information is then compared with (a machine representation of) the set of erroneous behaviours, where the definition of erroneous behaviour clearly depends on the nature of the analysis itself. If the intersection between those two sets is empty, the program is considered safe. Otherwise the analyser reports a warning. In this setting, the notions of soundness, completeness, false positives and false negatives can be better understood in terms of *overappriximation* and *underapproximation* of the program semantics.

A sound analysis computes an over approximation, that is a superset of the program semantics,

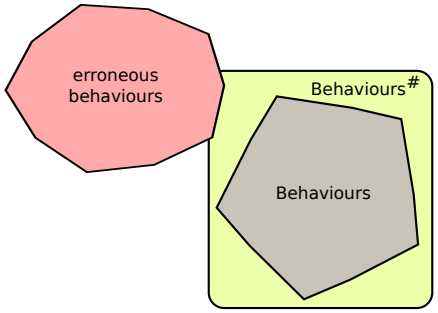


Figure 4.2: Overapproximation of the set of behaviours. The analysis reports a false positive since the set of abstract behaviours intersects the erroneous zone, while the actual program semantics doesn't.

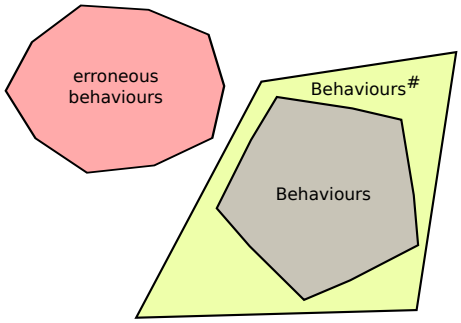


Figure 4.3: Overapproximation of the set of behaviours. The analysis successfully proved that the program is safe

as shown in figure 4.2. Here, the abstract semantics, $Behaviours^{\#}$, intersects the set of erroneous behaviours, and the analysis produces an alarm. However, it is in fact a false alarm, since the actual program semantics, $Behaviours$, is disjoint from the "erroneous zone". The analysis had in fact considered a *spurious behaviour*, a program behaviour that the program will never exhibit at runtime. Increasing precision can be understood as building a "smaller" approximation of $Behaviours$, such that it is still a superset of it but "small enough" (i.e. which contains less spurious behaviours) as to not intersect the dangerous zone. An example is shown in figure 4.3. In this case, since the approximation $Behaviours^{\#}$ is safe, and since $Behaviours^{\#}$ is actually a superset of $Behaviours$, the analysis has proved the program safe.

As seen before, precision needs to be traded off with scalability. Let us consider again figure 4.3. In this case, the approximation was still coarse enough to be easily computable (intuitively expressed by the simple shape of $Behaviours^{\#}$) yet it was precise enough to prove the property of

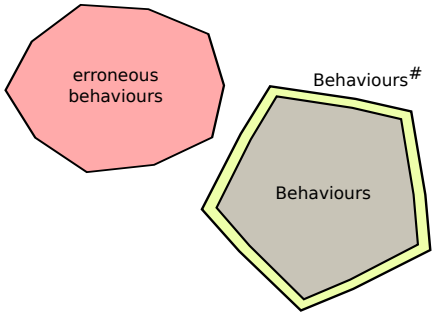


Figure 4.4: A "too precise" over approximation. The cost of a precise analysis can sometimes outweigh the benefits.

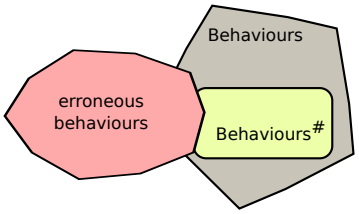


Figure 4.5: Underapproximation of the program semantics. The analysis finds some bugs (red area intersecting the set of analysed behaviours) but misses others (red area intersecting the actual program behaviours but missed by the analysis).

interest (no division-by-zero) without false alarms for *many different programs*. Intuitively, making the abstraction even more precise (as shown if figure 4.4) would make the analysis less efficient (at the point where it will become undecidable) while providing little advantage because the coarser abstraction of figure 4.3 was already "good enough" to prove safety for many programs. In other words, sometimes the cost of a much more precise abstraction outweight the benefits. The goal is then to design an abstraction which is coarse enough so that it can be easily computed while being precise enough so that the number of false positives can minimised for the particular property being analysed and the specific domain of application.

Dually, complete analyses can be understood as under approximations, that is subsets, of the program semantics. Figure 4.5 shows an example. Note that in this case, there is no possibility for false alarms, since $Behaviours^{\#}$ is a subset of $Behaviours$ which is the set of actual program behaviours. However, for the very same reason, the analysis might miss real bugs. In order to improve a complete analysis it is necessary to capture more behaviours. However, the more

behaviours are captured, the more the analysis will become inefficient and therefore less scalable. The same considerations about over-approximating analyses dually apply to under approximating analyses.

We conclude this part by stressing the fact that sound analyses are not necessarily *better* than unsound ones. Fully sound and fully complete analyses have in fact distinct applications. When the goal is to *prove* a program safe, then a sound analysis is appropriate, and the time spent dealing with false positives may be considered as a necessary evil. However, in less critical settings where the focus is on actually *finding* bugs, the fact that some bug is missed may be acceptable and actually more desirable then the presence of a large number of false positives. Ultimately, analyses might be not sound nor complete, instead being based on delicate tradeoffs between false positives and false negatives so as to achieve the sweet spot needed for that particular domain of application.

From here on we shall focus our discussion on sound analyses.

4.4 Static analysis vocabulary

In this section we introduce the basic static analysis terminology in a framework-independent way through examples. Later on we refine the intuitions in the context of the theory of abstract interpretation.

To get started, consider the following simple program consisting of a conditional statement where the actual value of the conditional `$cond` is statically unknown (the reader can think of this guard as an expression depending on some user input or random value):

```
if ($cond)
   $x = 1;
else
   $x = 2;
$y = 1 / $x;
```

We would like to determine wether there exists the possibility of a division-by-zero error at runtime.

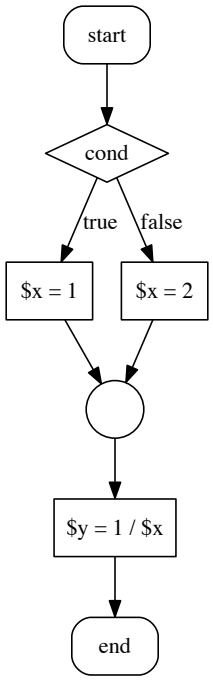


Figure 4.6: A simple CFG

More specifically, we are keen to determine wether the variable `$x` *can* be assigned the value zero and therefore produce a division-by-zero in the last line.

4.4.1 Exploring all paths

Before getting started, we introduce the *control flow graph* (CFG) representation for programs [93]. CFGs are widely used in compiler technology as well as in static analysis. The CFG of a program provides a compact representation of all of its possible execution paths. Figure 4.6 shows the CFG for the program above. Notice the presence of a circle-shaped node just after the two conditional branches. This kind of nodes are known as *junction* nodes. Note that the node has two entry edges (one for each branch of the preceding conditional) and one exit edge. This intuitively expresses the fact that, at junction nodes, different execution paths are *merged* into one. This concept will become clear in the following.

The goal of the analysis is to traverse all paths in the CFG and determine the set of all possible

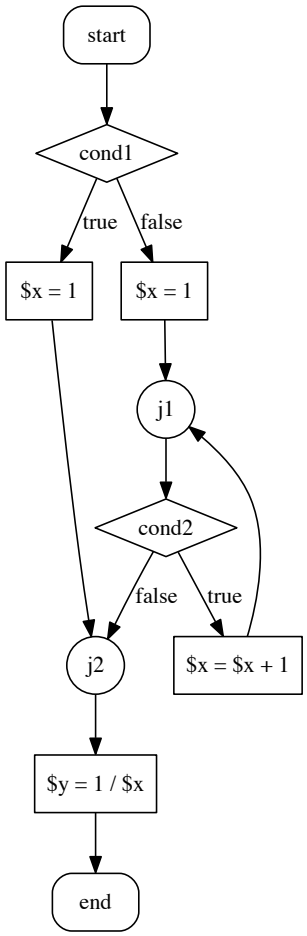


Figure 4.7: A slightly more involved CFG

values the variable `$x` might take at runtime. The simplest way to do this consists in traversing the paths while accumulating the assigned values for each assignment instruction into an *analysis environment* mapping variable names to *sets* of values. For the CFG in figure 4.6 there are only two paths to be explored. Let the analysis environment be initially initialised with $\texttt{\$x} : \emptyset$ If the `true` branch of the conditional is taken, then `$x` takes the value `1`, which is added to the set of possible values for `$x`. The environment is therefore updated to $\texttt{\$x} : \emptyset \cup \{1\}$. If the false branch is taken instead, then `$x` takes the value `2`, so it becomes $\$x \in \{1, 2\}$. Notice that *both* conditional branches are taken. Therefore, the analysis safely concludes that, since `$x` can only contain the values `1` or `2`, there can be no division-by-zero when assigning `$y` the result of `1 / $x`.

However this naive approach is not generally feasible. Programs might have an infinite number of paths or might not terminate, therefore some form of approximation is required in order to devise a computable analysis. To see that, consider a slightly more complicated example, where the conditions `$cond1` and `$cond2` are unknown and the CFG is shown if figure 4.7:

```
if ($cond1)
   $x = 1;
else {
   $x = 1
   while ($cond2)
      $x = $x + 1;
}
$y = 1 / $x;
```

The problem is that there is an infinite number of feasible paths in the program (one for each iteration of the `while` loop, since the guard is only known at runtime) as well as in infinite number of values the variable `x` could be assigned. Therefore, exploring all CFG paths one by one will lead to non-termination of the analysis.

4.4.2 Abstraction

We need to introduce the concept of *abstraction*. The idea is to keep enough information to be able to reason about the property of interest, while discarding the rest in order to make the analysis decidable and efficiently computable. In the context of our examples, we can observe how in fact, from the point of view of the property of interest, it doesn't matter wether the value of `$x` is `1`, `2` or `27`, because none of this value will cause a division-by-zero error. What really matters is wether `$x` can be zero or not.

We realise this intuitive idea by switching the domain of computation to an *abstract domain* (we discuss this idea more formally in section 4.5). One of the simplest abstract domains is the domain of *signs*. Essentially, the idea is to discard the actual value of a variable, keeping its sign

only. We have:

$$Signs = \{Pos, Neg, Zero\}$$

Therefore, for the purpose of our analysis, variables can only be assigned one of those three values. During the exploration of the CFG path, the encountered statements must be *simulated* so that they operate on those values only. For example, the statement `$x = 120` will cause the analysis state to be updated to `$x: Pos`, while the statement `$y = $x - 3` will lead to `$y: Unknown`, since not only `$x` contained the value `Pos`, but also the value `3` has been abstracted to `Pos`. Therefore, the assignment amounts to `$y = Pos - Pos`, i.e. all we know is that we are subtracting two positive numbers. Since it is not possible to know the sign of such an operation, `$y` is given the *conservative* value `unknown`, meaning that it can be either `Pos`, `Neg` or `Zero`. This is an example of information loss due to the abstraction.

To see how this approximation makes the analysis of our previous example computable, let's consider the following. If `cond1` is `true`, then the assignment `$x = 1` is executed. The analysis keeps track of this by updating the environment to $\$x : \bot \sqcup Pos$ (where, for now, $\bot$ and $\sqcup$ represent abstract versions of the empty set and set union operator respectively). Otherwise, we enter the `false` branch of the conditional, where `$x: Pos` because of the `$x = 1` assignment. Now the program *might* enter the `while` loop. Since the value of the guard `$cond2` is unknown, we don't know how many times the loop body will be executed, and therefore `$x` can potentially contain infinite values - this is what caused our previous analysis attempt to not terminate. However, if we keep only the sign of `$x`, the analysis will terminate. Consider the following. If the loop body is never executed, `$x` will keep its previous value, `Pos`. When the final assignment to `$y` is reached, we will have `$x: Pos`, and therefore the program will be considered safe. If the loop body is executed at least once, then the increment `$x = $x + 1` will be executed one or more times. However, since `$x` is positive (because of the previous assignments), and since adding two positive numbers yields a positive numbers, it will always be `$x: Pos`, no matter how many times the loop body is executed. This allows the analysis to terminate by detecting the fact that performing additional analysis steps does not change the accumulated analysis result. Formally, this means that the analysis has

reached a *fixpoint* (see 4.5). Once again, the program will be considered safe.

Let us now get back to the first example of figure 4.6 and change the code so that division-by-zero becomes possible:

```
if ($cond)
   $x = 0;
else
   $x = 1;
$y = 1 / $x;
```

According to our strategy, the possible values for `$x` will be $\{0, 1\}$ or `unknown` if using the domain of signs. In both cases, the analysis is able to catch the bug. In the concrete case, the presence of `0` in the set of possible values for `$x` is enough to raise an alarm. When using signs instead, the simulation of the two branches leads to `$x: Zero` and `$x: Pos` respectively, and therefore `pos` $\sqcup$ `Zero` = `Unknown`, which also causes an alarm to be triggered (since `Unknown` intuitively encodes all possible values, including zero). However, if we change the program again

```
if ($cond)
   $x = -1;
else
   $x = 1;
$y = 1 / $x;
```

it easy to see that our sign analysis will return a false alarm, since `Neg` $\sqcup$ `Pos` = `Unknown` but the only concrete values that variable `$x` may assume are `1` and `-1`. This loss of information is due to the nature of the domain itself. One way to improve the precision is therefore to adopt a more expressive domain. For example, instead of simply considering positive, negative or zero as the property of interest, we might enrich this by also adding a "non-zero" element where, for example, `Neg` $\sqcup$ `Pos` = `non-zero`. With this improved domain, the false alarm of the previous example would be avoided. However, as usual, more sophisticated domains generally lead to less scalable analyses.

4.4.3 Increasing precision with sensitivities

Let us consider an even simpler example such as the following:

```
$x = 1;
$x = 0;
$y = 1 / $x;
```

For this example, the simple analysis introduced above will compute $\{0, 1\}$ if accumulating concrete values or `Unknown` if computing signs. In both cases, it will successfully spot the division-by-zero error. Let us now swap the order of the first two assignments:

```
$x = 0
$x = 1
$y = 1 / $x;
```

Not surprisingly, our algorithm produces the same result and throws an alarm. But this time it is obvious that there is no possible division by zero, since the variable `$x`, that was initially zero, is reassigned with `1` just before being used as a denominator for the division operation.

Flow sensitivity

The analysis has produced a false alarm, and the reason for this is that the analysis strategy was *flow insensitive*: it does not take into account the order in which the statements are executed and the fact that the content of variables can *change over time*. For each variable, the analysis abstracts the set of values the variable might take during the execution of the whole program. The analysis reported a false alarm because the over approximation of the program semantics that it has computed contained a spurious execution in which `$x` is first assigned to `1` and then to `0`, where such an execution is clearly not possible at runtime.

In this cases, it is possible to enhance the analysis precision by adding *flow sensitivity*. Instead of keeping one global state that accumulates all possible values a variable can take during the whole program, we keep track of this information *for each program point*. In terms of CFG this amounts to annotating each edge of a CFG with an analysis environment (i.e. a mapping from variable

names to sets of values or abstract values) Figure 4.8 shows this approach in action with the

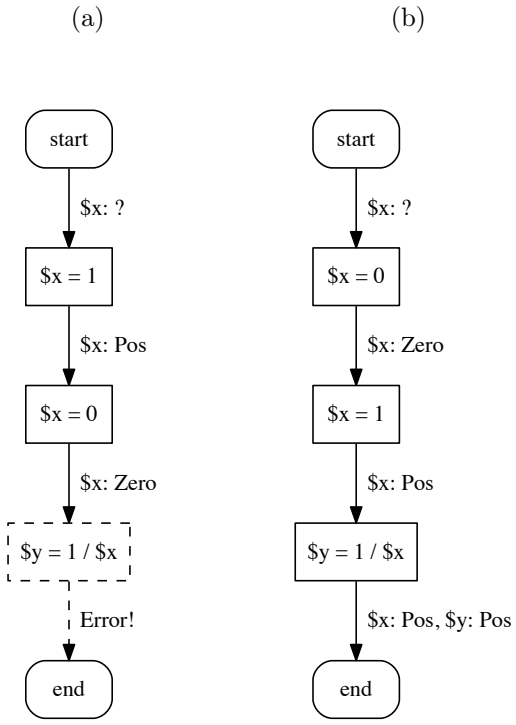


Figure 4.8: Using flow-sensitivity the analysis takes into account the order of execution of statements

two variants of the program above. The program whose CFG is shown in figure 4.8a contains a division-by-zero error which is successfully detected by the analysis. The program in 4.8b instead does not contain the error and is reported safe.

Path sensitivity

We introduce now another kind of sensitivity, *path sensitivity*. Consider the program

```
if ($z)
   $x = 1;
else
   $x = 0;
if ($z) then
   $y = 1 / $x;
```

Figure 4.9a shows the corresponding CFG annotated with the results of the flow sensitive analysis. The analysis concludes that in the program position immediately preceding the assignment `$y =`

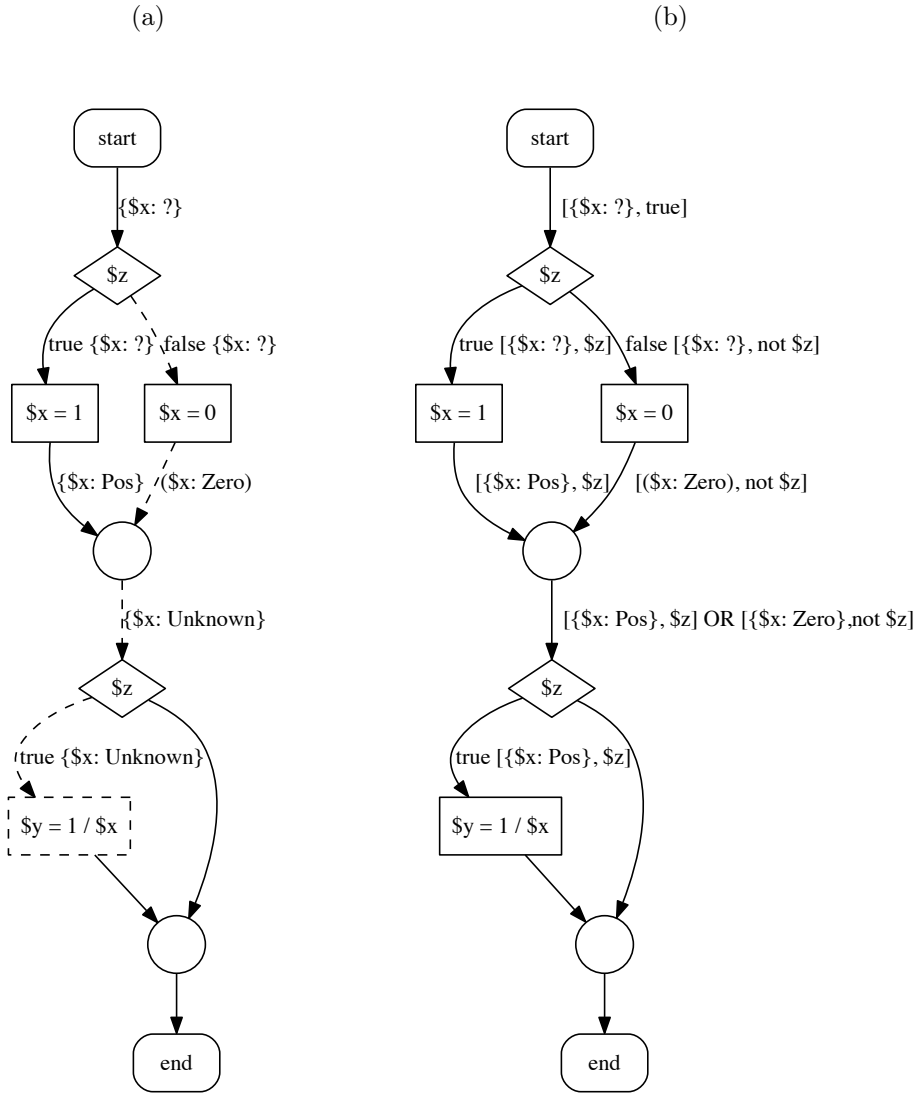


Figure 4.9: Adding path sensitivity in order to avoid infeasible paths

`1 / $x` the value of `$x` is `Unknown` and therefore raises an alarm. However, at closer inspection, it is clear that, while it is true that `$x` may contain the value zero during the execution of the program, there is actually no path that, when this happens, lead to the assignment containing the division. This is because, in this particular example, the two conditionals share the same guard `$z`. If `$z` is true, then `$x` gets assigned to `1` in the first conditional and then it gets used as a denominator for

dividing in the second conditional. On the other hand, if `$z` is false, `$x` gets the value zero, but the assignment operation in the second conditional will not be executed (since it is in the `true` branch of the conditional). Once again, the analysis reported a false alarm based on an infeasible path, i.e. a path that will never be realised at runtime. The infeasible path is shown with dashed edges in figure 4.9a.

In order to prune spurious paths we need to keep more information about the simulated program state. For example, instead of keeping only the signs (or desired property) of variables, we also keep track of the current *path condition*. Each time we enter a branch we keep track of the condition that allowed us to enter that branch as part of the state. Moreover, instead of merging multiple states into one at junction nodes, we keep them separated according to their path condition. Figure 4.9b shows this approach. By keeping track of the path conditions, the improved analysis detects the fact that in the program point immediately preceding the assignment involving the division can only be positive, and report the program as safe.

Context sensitivity

We conclude our overview of sensitivities by briefly discussing *context sensitivity*. Consider the following function that simply returns its argument

```
function id($x) {
   return $x;
}
```

and the following code

```
$a = id(0);
$b = id(1);
$c = 1 / $b;
```

The code above is clearly safe because the division is between `1` and the value contained in `$b` which is also `1`. However, if we apply our flow and path sensitive algorithm (even without applying any value abstraction) we get a false alarm as shown in figure 4.10a. The problem in this case is that the analysis does not match function calls with the corresponding returns, and for this reason it

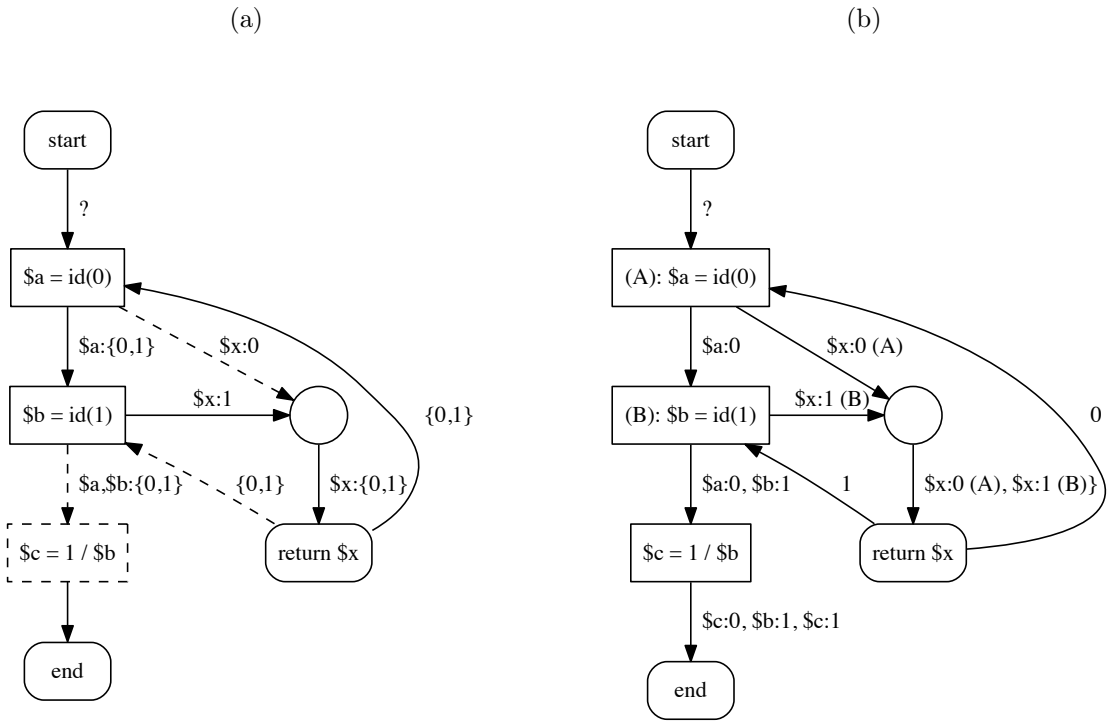


Figure 4.10: Adding context sensitivity

has considered an infeasible path where the return value of the first function call is returned to the second function call and therefore it is assigned to `$b`. Intuitively, one way of solving this consists in *labelling* each call site, for example by assigning a unique integer counter to each function call, and then keeping track of labels in each call and return edge on the CFG as in figure 4.10b. Alternatively, one could, instead of a single copy of the function's CFG, inline a copy of it into the original CFG for every call. Notice however, that this approach may introduce further challenges and complexity e.g. when dealing with looping constructs and recursive calls.

Are sensitivities always the best choice?

Adding sensitivities as shown in the previous examples increases the analysis precision and reduces the number of false positives. So one might ask why so many existing analyses lack some or all of the sensitivities. As discussed, there is a tradeoff between precision and scalability. Adding sensitivities improves precision by reducing scalability. When we added flow-sensitivity we had to

increase the amount of information to be tracked. We switched from one single global environment mapping each variable to its (abstract) value, to one such environment for *each* program point. When we added path sensitivity we further increased the complexity of the analysis by having to keep track of the path conditions and by avoiding collapsing two states into one at junction nodes, introducing possible issues of *combinatorial explosion*. Once again, finding the right combination of sensitivities for a particular analysis means finding the sweet spot between the expected efficiency of the analysis, the particular property to be handled, the expected dimension of the codebase, the expected use (i.e. prove programs safe or find bugs) and more.

4.5 A primer on Abstract Interpretation

The ideas discussed informally in the previous sections have been implemented and understood in many different, seemingly unrelated ways, including *type systems*, *data-flow analysis* and *constraint-based analysis* [93]. The theory of *abstract interpretation* [3] has been introduced in the late 70s by Patrick and Radia Cousot mainly as an attempt to provide a unifying theory to explain and understand the different approaches to static analysis and verification. Since then, the theory has been refined and extended and used to understand not only static analysis but also deductive verification, model checking and more, under a unified framework.

4.5.1 A bird's eye view

Say we want to compute the semantics $S[\![P]\!]$ of a program P (e.g. set of reachable states, set of returned values, etc.). Let us call $S[\![P]\!]$ the *concrete semantics*. Depending on the domain of application, $S[\![P]\!]$, can be understood as an element of some algebraic structure $\mathcal{C}$, known as the *concrete domain* (e.g. sets of states, sets of values, sets of execution traces etc.). While, in general, $S[\![P]\!]$ has a well defined mathematical specification (usually as the solution of a fixpoint equation), it is usually non computable due to undecidability or complexity (as extensively discussed before).

The idea behind abstract interpretation is that we can still compute an approximation (i.e.

a property) $S^{\#}[\![P]\!]$ of $S[\![P]\!]$ in order to answer a *specific* question - instead of computing an exact solution to our problem (such as the exact set of reachable states or returned values), we only compute a *property* of it. For example, instead of computing the exact set of values possibly returned by an arithmetic expression when evaluated in all possible execution environments, we may only calculate a property of the whole set, such as "all returned values are positive". We obviously loose information during the process (i.e we don't know the *exact* set of values or states), but the insight is that, given a specific question to be answered, the approximate information alone, if chosen accordingly, is enough to answer the question. For example, establishing that during execution of program P all variables are non-zero, is enough to prove that no division-by-zero can occur at runtime. However notice that given a different question to be answered, e.g. "are all array accesses in-bound?" a different abstraction may be needed.

Suppose then we identified the exact property we are interested in assessing about $S[\![P]\!]$. The first step consists in designing an *abstract domain*, $\mathcal{A}$ that encodes the property of interest, together with a pair of functions $\alpha : \mathcal{C} \to \mathcal{A}$, for *abstraction*, and $\gamma : \mathcal{A} \to \mathcal{C}$, for *concretisation*. Provided α and γ enjoy a certain set of mathematical properties (which we formally define in the following), it is then possible to *systematically derive* a specification of $S^{\#}[\![P]\!]$ by

- Replacing $\mathcal{C}$ with $\mathcal{A}$ in the definition of S
- replacing every function $f : \mathcal{C}^n \to \mathcal{C}(n = 1, 2, \ldots)$ used in the definition of S with a corresponding function $f^{\#} : \mathcal{A}^n \to \mathcal{A}$ which "correctly mimics" the behaviour of f in $\mathcal{A}$.

The derived abstract semantics $S^{\#}[\![P]\!]$ is guaranteed to be a *correct approximation* of $S[\![P]\!]$, in the sense that the property encoded by $S^{\#}[\![P]\!]$ (e.g. "no division by zero can occur" or "no tainted data flows into a sink") is satisfied by $S[\![P]\!]$. If the approximation is coarse enough, $S^{\#}$ is *effectively computable* and can be straightforwardly translated into an analyser. In the next subsection we put this concept into practice by showing how we derive an abstract interpreter for a simple language of arithmetic expressions with variables, inspired by [94]. While simple, the choosen example gives an exact idea of the workflow involved when systematically designing an abstract interpreter, and it inspired our own work.

```
Numbers: n, m, ...
Variables ::= X, Y, ...
AExp ::= n
       | X
       | rand
       | + AExp | - AExp
       | AExp + AExp | AExp * AExp
```

Figure 4.11: Syntax of the language of arithmetic expressions

4.5.2 Case study: abstract interpretation of arithmetic expressions

Machine arithmetic

Let $\mathbb{I}$ be the set of machine integers. In order to keep the exposition simple, we make the simplistic assumption $\mathbb{I} = \mathbb{Z}$ (where $\mathbb{Z}$ is intended, as usual, as the set of integers). A more realistic treatment which takes into account overflow errors can be found in [94]. Given $n \in \mathbb{Z}$ we denote its machine-arithmetic counterpart with $\bar{n} \in \mathbb{I}$. Similarly, given unary and binary mathematical operations $u : \mathbb{Z} \to \mathbb{Z}$ and $b : \mathbb{Z}^2 \to \mathbb{Z}$ we write $\bar{u} : \mathbb{I} \to \mathbb{I}$ and $\bar{b} : \mathbb{I}^2 \to \mathbb{I}$ for their corresponding machine arithmetic operations.

Environments

We define program states ρ as *environments* mapping variables to machine values

$$\rho \in Env : Var \to \mathbb{I}$$

such that the value of a variable X can be obtained as $\rho(X)$.

Syntax and semantics of arithmetic expressions

The syntax of our language is shown if figure 4.11. To give semantics to the language, we provide a set of simple big-step judgments in the form

$$\rho \vdash A \Downarrow v$$

$$\rho \vdash n \Downarrow \bar{n} \qquad \text{constant}$$

$$\rho \vdash X \Downarrow \rho(X) \qquad \text{variable}$$

$$\frac{i \in \mathbb{I}}{\rho \vdash rand \Downarrow i} \qquad \text{random value}$$

$$\frac{\rho \vdash A \Downarrow v}{\rho \vdash uA \Downarrow \bar{u}v} \qquad \text{unary operators}$$

$$\frac{\rho \vdash A_1 \Downarrow v_1 \wedge \rho \vdash A_2 \Downarrow v_2}{\rho \vdash A_1 b A_2 \Downarrow v_1 \bar{b} v_2} \qquad \text{binary operators}$$

Figure 4.12: Big-step operational semantic of a simple language of arithmetic expressions.

meaning that in environment ρ the arithmetic expression A may evaluate to a value $v \in \mathbb{I}$. The semantics of our language is then completely defined by the five rules shown in figure 4.12.

Concrete semantics

We now turn to the problem of *analysing* arithmetic expressions. Given an arithmetic expression A and a *set of* environments $R \subseteq Env$, we define the *concrete collecting semantics* of A as the set of all possible values the expression can evaluate to when evaluated in any environment $\rho \in R$

$$collect : AExp \to \mathcal{P}(Env) \to \mathcal{P}(\mathbb{I})$$

$$collect[\![A]\!]R = \{v \in \mathbb{I} | \exists \rho \in R. \rho \vdash A \Downarrow v\}$$

noting that when $R = Env$ then $collect[\![A]\!]R$ effectively defines the set of *all* possible values A can evaluate to in *all* possible execution environments. As discussed before, *collect* is not computable and therefore an approximation will be needed.

Concrete domain: properties of objects

A value property is understood as the set of values which have this property. The concrete properties of values are therefore elements of the powerset $\mathcal{P}(\mathbb{I})$. For example $\{1, 2, \ldots\} \in \mathcal{P}(\mathbb{I})$ is

the property "is a positive machine integer" while $\{2n+1|n=1,2,\ldots\} \in \mathcal{P}(\mathbb{I})$ is the property "is an odd machine integer". As seen, the collecting semantics $collect[\![A]\!]R$ is an element of $\mathcal{P}(\mathbb{I})$. Therefore the set of value properties $\mathcal{P}(\mathbb{I})$ is our concrete domain - the universe of objects on which the semantics is computed, and $collect[\![A]\!]R$ specifies the strongest property that values of A satisfy when evaluated in an environment $\rho \in R$.

$(\mathcal{P}(\mathbb{I}), \subseteq, \emptyset, \mathcal{P}(\mathbb{I}), \cup, \cap, \neg)$ is a complete boolean lattice [94]. Elements of the $powerset\mathcal{P}(\mathbb{I})$ are understood as properties of values with subset inclusion $\subseteq$ as logical implication, $\emptyset$ as false, $\mathcal{P}(\mathbb{I})$ as true, $\cup$ as the disjunction, $\cap$ as the conjunction and $\neg$ as the negation.

Galois connection based abstraction

We have seen that computing $collect[\![A]\!]R \in \mathcal{P}(\mathbb{I})$ is undecidable. In general, since computers are finite machines, we can only use a machine encoding $\mathcal{A}$ of a subset of all possible value properties, i.e. a set of *abstract properties*. We call $\mathcal{A}$ an *abstract domain*. Let's assume $(\mathcal{A}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a complete lattice. As in $\mathcal{P}(\mathbb{I})$, $\sqsubseteq$ (known as the *approximation ordering*) encodes logical implication, $\bot$ encodes false, $\top$ encodes true and $\sqcup$ and $\sqcap$ encodes disjunction and conjunction, respectively. The meaning of each $a \in \mathcal{A}$ is given by a concretisation function

$$\gamma : \mathcal{A} \to \mathcal{P}(\mathbb{I})$$

which maps abstract properties into concrete ones. In order for $\sqsubseteq$ to encode implication, we require γ to be monotonic

$$a \sqsubseteq b \implies \gamma(a) \subseteq \gamma(b) \tag{4.1}$$

Given an arbitrary concrete value property $P \in \mathcal{P}(\mathbb{I})$, P has in general no *exact* counterpart in $\mathcal{A}$. However, P can be over-approximated by any $p \in \mathcal{A}$ such that

$$P \subseteq \gamma(\bar{p})$$

In this case, we say that p over-approximates P in the abstract domain $\mathcal{A}$. While, in principle, a concrete property P can be over-approximated by more than one element of $\mathcal{A}$, we now turn our attention to the concept of *best approximation*. Given $P \in \mathcal{P}(\mathbb{I})$ we now define an *abstraction function*

$$\alpha : \mathcal{P}(\mathbb{I}) \to \mathcal{A}$$

in such a way that $\alpha(P)$ encodes the "best approximation" of P in $\mathcal{A}$. Formally, this situation is that of a *Galois Connection* [94, 3]. The abstraction function α must satisfy the following set of axioms:

$$P \subseteq Q \implies \alpha(P) \sqsubseteq \alpha(Q) \qquad \alpha \text{ is monotone} \tag{4.2}$$

$$\forall P \in \mathcal{P}(\mathbb{I}), P \subseteq \gamma(\alpha(P)) \qquad \alpha(P) \text{ overapproximates P} \tag{4.3}$$

$$\forall p \in \mathcal{A}, \alpha(\gamma(p)) \sqsubseteq p \qquad \text{the composition } \alpha \circ \gamma \text{ is reductive} \tag{4.4}$$

$$\tag{4.5}$$

Assume now that α satisfies the axioms above. Consider a concrete property P and an over approximation $p \in \mathcal{A}$, i.e. $P \subseteq \gamma(p)$. Then, from the monotonicity of α it follows that

$$\alpha(P) \sqsubseteq \alpha(\gamma(p))$$

Since the composition $\alpha \circ \gamma$ is reductive it follows

$$\alpha(P) \sqsubseteq p$$

or, considering meanings (4.1),

$$\gamma(\alpha(P)) \subseteq \gamma(p)$$

expressing the fact that $\alpha(P)$ encodes the best possible approximation of P in $\mathcal{A}$. When conditions 4.1, 4.2, 4.3 and 4.4 are met, we say that the lattices $\mathcal{P}(\mathbb{I})$ and $\mathcal{A}$ are in Galois connection, denoted with

$$\langle \mathcal{P}(\mathbb{I}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq \rangle \tag{4.6}$$

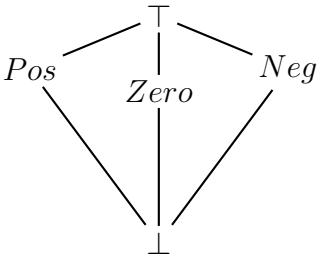
Galois connections formalise the everyday concept of approximation.

An abstract domain for arithmetic expressions: The *signs* domain

We can now design an abstract domain to approximate the semantics of arithmetic expressions. Suppose we are interested in determining the *sign* of arithmetic expressions. Then we design an abstract domain such as

$$\mathcal{A} = Signs = \{\bot, Pos, Zero, Neg, \top\}$$

with the approximation ordering defined as in the *Hasse diagram*



A possible concretisation function is

$$\gamma(\bot) = \emptyset$$

$$\gamma(\top) = \mathcal{P}(\mathbb{I})$$

$$\gamma(Pos) = \{1, 2, \ldots\} \in \mathcal{P}(\mathbb{I})$$

$$\gamma(Zero) = \{0\} \in \mathcal{P}(\mathbb{I})$$

$$\gamma(Neg) = \{\ldots, -2, -1\} \in \mathcal{P}(\mathbb{I})$$

and a possible abstraction function is

$$\alpha(P) = \begin{cases} \alpha(P) = \bot & \text{when } P = \emptyset \\ \alpha(P) = Pos & \text{when } P \subseteq \{1, 2, \ldots\} \\ \alpha(P) = Zero & \text{when } P = \{0\} \\ \alpha(P) = Neg & \text{when } P \subseteq \{\ldots, -2, -1\} \\ \alpha(P) = \top & \text{otherwise} \end{cases}$$

It can be proven that α and γ as given above form a Galois connection, therefore

$$\langle \mathcal{P}(\mathbb{I}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Signs, \sqsubseteq \rangle \tag{4.7}$$

Approximating environments

Recall that

$$collect : AExp \to \mathcal{P}(Env) \to \mathcal{P}(\mathbb{I})$$

i.e. the collecting semantics of an arithmetic axpression A takes a *set* of environments as input and returns a set of values. In the previous part we have designed an approximation of $\mathcal{P}(\mathbb{I})$ by providing a lattice $Sign$ of signs and functions α and γ so as to form a Galois connection. Our final goal will be to derive an approximation of *collect*, say $collect^{\#}$ such that

$$collect^{\#} : AExp \to X \to Signs$$

However, notice that the domain X *cannot* be $\mathcal{P}(Env)$ as in *collect*, for similar undecidability and complexity arguments. It follows that, in order to derive a *computable* approximation of *collect*, we must also provide an approximation of *sets of environments*.

Similarly to properties of values $\mathcal{P}(\mathbb{I})$, environment properties are also understood as the set of environments which have that property. For example, the property "being environments where $X = Y$" can be expressed as the set

$$\{\rho \in Env | \rho(X) = \rho(Y)\}$$

A common way to abstract sets of environments consist in first performing a non-relational abstraction such that

$$\langle \mathcal{P}(Var \to \mathbb{I}), \subseteq \rangle \underset{\alpha_r}{\overset{\gamma_r}{\leftrightarrows}} \langle Var \to \mathcal{P}(\mathbb{I}), \dot{\subseteq} \rangle \tag{4.8}$$

where

$$\alpha_r(R) = \lambda X \in Var.\{\rho(X) | \rho \in R\}$$
$$\gamma_r(r) = \{\rho | \forall X \in Var : \rho(X) \in r(X)\}$$

and

$$r_1 \dot{\subseteq} r_2 \iff \forall X \in Var : r_1(X) \subseteq r_2(X)$$

Consider for example the set (or property) of environments

$$R = \{[x \mapsto 1, y \mapsto 2], [x \mapsto 2, y \mapsto 1]\} \in \mathcal{P}(Env)$$

we have

$$\alpha(R) = [x \mapsto \{1, 2\}1, y \mapsto \{1, 2\}] \in Var \to \mathcal{P}(\mathbb{I})$$

and

$$\gamma([x \mapsto \{1, 2\}1, y \mapsto \{1, 2\}]) =$$
$$\{[x \mapsto 1, y \mapsto 1], [x \mapsto 1, y \mapsto 2], [x \mapsto 1, y \mapsto 2], [x \mapsto 2, y \mapsto 1]\} \in \mathcal{P}(Env)$$

Notice how the chosen abstraction process caused the loss of information regarding the *relationship* between variables.

The next step consists in using the existing approximation

$$\langle \mathcal{P}(\mathbb{I}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Signs, \sqsubseteq \rangle$$

in order to approximate the codomain

$$\langle Var \to \mathcal{P}(\mathbb{I}), \dot{\subseteq} \rangle \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} \langle Var \to Signs, \dot{\sqsubseteq} \rangle \tag{4.9}$$

as follows

$$r_1 \dot{\sqsubseteq} r_2 \iff \forall X \in Var : r_1(X) \sqsubseteq r_2(X)$$

$$\alpha_c(R) = \alpha \circ R$$

$$\gamma_c(r) = \gamma \circ r$$

Since the composition of Galois connections

$$\langle D_1, \sqsubseteq_1 \rangle \underset{\alpha_{1,2}}{\overset{\gamma_{2,1}}{\leftrightarrows}} \langle D_2, \sqsubseteq_2 \rangle \text{ and } \langle D_2, \sqsubseteq_1 \rangle \underset{\alpha_{2,3}}{\overset{\gamma_{3,2}}{\leftrightarrows}} \langle D_3, \sqsubseteq_2 \rangle$$

is a Galois connection [94]

$$\langle D_1, \sqsubseteq_1 \rangle \underset{\alpha_{2,3} \circ \alpha_{1,2}}{\overset{\gamma_{2,1} \circ \gamma_{3,2}}{\leftrightarrows}} \langle D_3, \sqsubseteq_3 \rangle$$

then we conclude

$$\langle \mathcal{P}(Var \to \mathbb{I}), \subseteq \rangle \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \langle Var \to Signs, \dot{\sqsubseteq} \rangle \tag{4.10}$$

with

$$\dot{\alpha}(R) = \alpha_c \circ \alpha_r(R) = \lambda X \in Var.\alpha(\{\rho(X) | \rho \in R\})$$

$$\dot{\gamma}(r) = \gamma_r \circ \gamma_c(r) = \{\rho | \forall X \in Var.\rho(X) \in \gamma(r(X))\}$$

For convenience, we denote with $Env^{\#} = Var \to Signs$ the set of abstract environments.

Abstract semantics of arithmetic expressions

We now have all the ingredients necessary to *derive* an abstract semantics

$$collect^{\#} : AExp \to Env^{\#} \to Signs$$

which correctly approximates *collect* in the sense that

$$\forall \rho^{\#} \in Env^{\#}, A \in Exp, collect[\![A]\!]\gamma(\rho^{\#}) \subseteq \gamma(collect^{\#}[\![A]\!]\rho^{\#})$$

To do that, we start with the *functional abstraction* [95, 94]:

$$\langle \mathbb{P}(Var \to \mathbb{I}) \to \mathcal{P}(\mathbb{I}), \dot{\subseteq} \rangle \underset{\alpha^{\triangleright}}{\overset{\gamma^{\triangleright}}{\leftrightarrows}} \langle (Var \to Signs) \to Signs, \dot{\sqsubseteq} \rangle \tag{4.11}$$

where

$$\alpha^{\triangleright}(\Phi) = \alpha \circ \Phi \circ \dot{\gamma} \qquad\qquad \gamma^{\triangleright}(\phi) = \gamma \circ \phi \circ \dot{\alpha}$$

Intuitively, our goal is to derive a function $\phi : (Var \to Signs) \to Signs$ which, for any abstract environment $p \in (Var \to Signs)$ provides an *overestimate* $\phi(p)$ of the concrete set of outcomes $\Phi(\gamma(p))$ provided by the concrete semantics Φ. This soundness requirement can be formalised as follows

$$\forall p \in Env^{\#} : \Phi(\dot{\gamma}(p)) \subseteq \gamma(\phi(p))$$

For the monotony of α, the fact that the composition $\alpha \circ \gamma$ is reductive and by definition of $\alpha^{\triangleright}$ we get

$$
\begin{aligned}
&\forall p \in Env^{\#} : \Phi(\dot{\gamma}(p)) \subseteq \gamma(\phi(p)) \\
\Longleftrightarrow &\forall p \in Env^{\#} : \alpha(\Phi(\dot{\gamma}(p))) \sqsubseteq \alpha(\gamma(\phi(p))) \sqsubseteq \phi(p) \\
\Longleftrightarrow &\alpha \circ \Phi \circ \dot{\gamma} \mathrel{\dot{\sqsubseteq}} \phi \\
\Longleftrightarrow &\alpha^{\triangleright}(\Phi) \mathrel{\dot{\sqsubseteq}} \phi
\end{aligned}
$$

expressing the fact that $\alpha^{\triangleright}(\Phi)$ is the the best possible sound approximation of Φ. It is important to note that since Φ and α are in general not computable, $\alpha^{\triangleright}(\Phi)$ can only be used as a *formal specification*, as it needs to be further refined in order to obtain an effectively computable function or algorithm.

We now show how to derive an abstract semantics for arithmetic expressions $collect^{\#} : Env^{\#} \to Signs$ as an *approximation* of the best possible abstraction $\alpha^{\triangleright}(collect)$ of the concrete semantics $collect : \mathbb{P}(Env) \to \mathcal{P}(\mathbb{I})$ in such a way that

$$
\alpha^{\triangleright}(collect)[\![A]\!] \mathrel{\dot{\sqsubseteq}} collect^{\#}[\![A]\!] \tag{4.12}
$$

We start from the formal specification $\alpha^{\triangleright}(collect[\![A]\!])$

$$
\begin{aligned}
&\alpha^{\triangleright}(collect[\![A]\!]) \\
&= \alpha \circ collect[\![A]\!] \circ \dot{\gamma} && \text{(definition of } \alpha^{\triangleright}) \\
&= \lambda r.\alpha(collect[\![A]\!](\dot{\gamma}(r))) && \text{(definition of function composition } \alpha^{\triangleright}) \\
&= \lambda r.\alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash A \Downarrow v\}) && \text{(definition of } collect)
\end{aligned}
$$

If r is the lattice infimum $\lambda Y.\bot$ and $\gamma(\bot) = \emptyset$ we obtain

$$
\begin{aligned}
&\alpha^{\triangleright}(\mathit{collect}[\![A]\!])(\lambda Y.\bot) \\
&= \alpha(\emptyset) && \text{(definition of } \dot{\gamma}) \\
&= \emptyset && \text{(Galois connection)}
\end{aligned}
$$

Otherwise (i.e. when $r \neq \lambda Y.\bot$ or $\gamma(\bot) \neq \emptyset$),

$$
\begin{aligned}
&\alpha^{\triangleright}(\mathit{collect}[\![A]\!])r \\
&= \lambda r.\alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash A \Downarrow v\})r \\
&= \alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash A \Downarrow v\}) && \text{(definition of lambda expressions)}
\end{aligned}
$$

We can now proceed by induction on the syntactic structure of the arithmetic expression A. Let us first consider the simple case where $A = n \in \mathbb{Z}$

$$
\begin{aligned}
&\alpha^{\triangleright}(\mathit{collect}[\![n]\!])r \\
&= \alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash n \Downarrow v\}) \\
&= \alpha(\{n\}) && \text{(definition of } \rho \vdash n \Downarrow v) \\
&= \mathit{collect}^{\#}[\![n]\!]r && \text{(by defining } \mathit{collect}^{\#}[\![n]\!]r = \alpha(\{n\}))
\end{aligned}
$$

What we just derived is intuitive. The abstract interpretation of a (constant) arithmetic expression is its sign as given by the abstraction function α. Notice that, although α might not be computable in general, in this case what is required is only to compute the abstraction of the singleton $\{n\}$, which is usually a simpler problem (e.g. for our sign analysis it can be implemented by a simple conditional assigning the abstract value *Pos* is the argument is positive etc.).

Let us now consider the case where the arithmetic expression A is a variable.

$$
\begin{aligned}
&\alpha^{\triangleright}(\mathit{collect}[\![X]\!])r \\
&= \alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash X \Downarrow v\}) \\
&= \alpha(\{\rho(X) | \rho \in \dot{\gamma}(r)\}) && \text{(definition of } \rho \vdash X \Downarrow v) \\
&= \alpha(\gamma(r(X))) && \text{(definition of } \dot{\gamma}) \\
&\sqsubseteq r(X) && \text{(Galois connection)} \\
&= \mathit{collect}^{\#}[\![X]\!]r && \text{(by defining } \mathit{collect}^{\#}[\![X]\!]r = r(X))
\end{aligned}
$$

Therefore, a variable is evaluated by reading its content from the abstract environment. Notice how, formally, $\alpha^{\triangleright}(\mathit{collect}[\![X]\!])r \sqsubseteq \mathit{collect}^{\#}[\![X]\!]r$, i.e. the abstract semantics we are designing is an approximation of its formal specification as given by the functional abstraction $\alpha^{\triangleright}$.

We shall now consider the case of binary operations. Consider $A = A_1 b A_2$, with $b \in \{+, -, /, \mathit{mod}, ...\}$ (the following calculation is independent of the particular binary operator)

$$
\begin{aligned}
&\alpha^{\triangleright}(\mathit{collect}[\![A_1 b A_2]\!])r \\
&= \alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_1 b A_2 \Downarrow v\}) \\
&= \alpha(\{v_1 \bar{b} v_2 | \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_1 \Downarrow v_1 \wedge \rho \vdash A_2 \Downarrow v_2\}) \\
&\qquad \text{(definition of } \rho \vdash A_1 b A_2 \Downarrow v) \\
&\sqsubseteq \alpha(\{v_1 \bar{b} v_2 | \exists \rho_1 \in \dot{\gamma}(r) : \rho \vdash A_1 \Downarrow v_1 \wedge \exists \rho_2 \in \dot{\gamma}(r) : \rho_2 \vdash A_2 \Downarrow v_2\}) \\
&\qquad (\alpha \text{ monotone}) \\
&\sqsubseteq \alpha(\{v_1 \bar{b} v_2 | v_1 \in \gamma \circ \alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_1 \Downarrow v\}) \wedge \\
&\qquad\qquad v_2 \in \gamma \circ \alpha(\{v | \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_1 \Downarrow v\}))) \\
&\qquad (\alpha \text{ monotone}, \gamma \circ \alpha \text{ extensive}) \\
&\sqsubseteq \alpha(\{v_1 \bar{b} v_2 | v_1 \in \gamma(\mathit{collect}[\![A_1]\!])r) \wedge v_2 \in \gamma(\mathit{collect}[\![A_2]\!])r)\}) \\
&\qquad \text{(induction hypothesis 4.12)} \\
&\sqsubseteq b^{\triangleright}(\mathit{collect}[\![A_1]\!]r, \mathit{collect}[\![A_2]\!]r) \\
&\qquad \text{(by defining } b^{\triangleright} \text{ so that } \alpha(\{v_1 \bar{b} v2 | v_1 \in \gamma(p_1) \wedge v_2 \in \gamma(p_1)\}) \sqsubseteq b^{\triangleright}(p_1, p_2) \text{)} \\
&= \mathit{collect}^{\#}[\![A_1 b A_2]\!]r \\
&\qquad \text{(by defining } \mathit{collect}^{\#}[\![A_1 b A_2]\!]r = b^{\triangleright}(\mathit{collect}[\![A_1]\!]r, \mathit{collect}[\![A_2]\!]r))
\end{aligned}
$$

The cases for unary operators and the *rand* operations are derived with similar techniques, and we omit them for brevity (the interested reader is referred to [94]). In conclusion, we have designed an abstract semantics $\mathit{collect}^{\#}$ which satisfies the soundness requirement 4.12. The semantics, whose structure closely resembles a standard functional interpreter, is summarised in figure 4.13

Abstract arithmetic operations

As a last step, we must provide abstract operations $?^{\triangleright}$, $u^{\triangleright}$ and $b^{\triangleright}$ so that they satisfy the soundness requirement as in figure 4.13. We only show a few cases. For $?^{\triangleright}$, since

$$
\begin{array}{lll}
collect^{\#}[\![A]\!](\lambda Y.\bot) & =\bot & \text{if } \gamma(\bot) = \emptyset \\
collect^{\#}[\![n]\!]r & =\alpha(\{n\}) & \\
collect^{\#}[\![X]\!]r & =r(X) & \\
collect^{\#}[\![rand]\!]r & =?^{\triangleright} & \\
collect^{\#}[\![uA_1]\!]r & =u^{\triangleright}(collect^{\#}[\![A_1]\!]r) & \\
collect^{\#}[\![A_1 b A_2]\!]r & =b^{\triangleright}(collect^{\#}[\![A_1]\!]r, collect^{\#}[\![A_2]\!]r) &
\end{array}
$$

parametrised by the abstract operations

$?^{\triangleright} \sqsupseteq \alpha(\mathbb{I})$
$u^{\triangleright}(p) \sqsupseteq \alpha(\{\bar{u}v | v \in \gamma(p)\})$
$b^{\triangleright}(p_1, p_2) \sqsupseteq \alpha(\{v_1 \bar{b} v_2 | v_1 \in \gamma(p_1) \wedge v_2 \in \gamma(p_1)\})$

Figure 4.13: Abstract semantics of the language of arithmetic expressions

$$\alpha(\mathbb{I}) = \top$$

we simply define

$$?^{\triangleright} = \top$$

which clearly satisfies the requirement

$$?^{\triangleright} \sqsupseteq \alpha(\mathbb{I}) = \top$$

Intuitively, since the *rand* operator can return any number, we conservatively abstract it into an operator which returns *Top*, i.e. *any* number.

Let us consider binary operator, say the addition operator. It is required that

$$+^{\triangleright}(p_1, p_2) \sqsupseteq \alpha(\{v_1 + v_2 | v_1 \in \gamma(p_1) \wedge v_2 \in \gamma(p_1)\})$$

The design proceeds by case analysis on the arguments p_1 and p_2. Consider for example the case where $p_1 = Pos$ and $p_2 = Pos$. Then

$$
\begin{aligned}
&+^{\triangleright}(Pos, Pos) \sqsupseteq \alpha(\{v_1 + v_2 | v_1 \in \gamma(Pos) \wedge v_2 \in \gamma(Pos)\}) \\
&= \alpha(\{v_1 + v_2 | v_1 \in \{1, 2, \ldots\} \wedge v_2 \in \{1, 2, \ldots\}\}) \\
&= \alpha(\{2, 3, \ldots\}) \\
&= Pos
\end{aligned}
$$

therefore we define $+^{\triangleright}(Pos, Pos) = Pos$, expressing the well-known fact that the sum of two positive numbers is itself a positive number. Consider instead the case $p_1 = Pos$ and $p_2 = Neg$. We obtain

$$
\begin{aligned}
&+^{\triangleright}(Pos, Neg) \sqsupseteq \alpha(\{v_1 + v_2 | v_1 \in \gamma(Pos) \wedge v_2 \in \gamma(Neg)\}) \\
&= \alpha(\{v_1 + v_2 | v_1 \in \{1, 2, \ldots\} \wedge v_2 \in \{\ldots, -2, -1\}\}) \\
&= \alpha(\{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\
&= \top
\end{aligned}
$$

Notice how, in this case, we loose precision, since the sign of the sum of a positive and a negative number is not known and therefore must be conservatively approximated to $\top$. In fact, the obtained rules correspond to the usual *rule of signs* for arithmetics. The complete definition of $+^{\triangleright}$ is given in the following table:

$+^{\triangleright}$	$\bot$	Pos	$Zero$	Neg	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
Pos	$\bot$	Pos	Pos	$\top$	$\top$
$Zero$	$\bot$	Pos	$Zero$	Neg	$\top$
Neg	$\bot$	$\top$	Neg	Neg	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

The other abstract operations (e.g. $*^{\triangleright}$ and $mod^{\triangleright}$) are designed in a similar way.

As a concluding remark, notice how the abstract semantics $collect^{\#}$ (as defined in figure 4.13) is *parametric* in the abstract domain and abstract arithmetic operations. This means that the obtained abstract interpreter (or analyser) can be adapted to calculate different properties (such add parity, for example), simply by providing a different domain together with an appropriate set of operations. In terms of engineering this paves the way to the design of modular abstract interpreters, where different analyses are obtained simply by including (or importing) a different module defining the domain of interest, where the domain-independent parts of the interpreter does not need to be redesigned.

4.5.3 Fixpoint abstraction

Our proposed case study was particularly simple yet we believe it gives a rough idea of the principles involved in the systematic abstraction of a programming language or calculus equipped with a formal semantics. One notable omission, which is also an important part of abstract interpretation theory and practice, is the concept of *fixpoint abstraction*, which we briefly outline here.

First, recall that given a monotonic function $f : L \to L$ over a lattice L with ordering $\sqsubseteq$ and bottom element $\bot$, the *least fixed point* $lfp^{\sqsubseteq}_{m} f$ is characterised by:

$$f(lfp^{\sqsubseteq}_{\bot} f) = lfp^{\sqsubseteq}_{\bot} f \tag{4.13}$$

$$(f(x) = x) \Rightarrow lfp^{\sqsubseteq}_{\bot} f \sqsubseteq x \tag{4.14}$$

In words, $lfp^{\sqsubseteq}_{\bot} f$ is the smallest element y of L (according to the ordering $\sqsubseteq$) for which $f(y) = y$. For ease of notation, we shall simply write $lfp\ f$ instead of $lfp^{\sqsubseteq}_{\bot} f$, when this is not ambiguous.

In general, the semantics of (imperative) programming languages (sets of reachable states, sets of traces) is in fact defined as the least fixed point of some monotone function F over a concrete domain $\mathcal{C}$ (say sets of states for example). In this case, provided one has applied the principles outlined before, an approximation of the semantics can often be computed as the least fixed point

of a function $F^{\#}$ over the abstract domain $\mathcal{A}$. The soundness of the approach is given by the *fixpoint transfer theorem*, which guarantees that the fixed point of the abstract function $F^{\#}$ is a sound approximation of the actual fixed point of F. Formally,

$$\alpha(lfp\ F) \sqsubseteq lfp\ F^{\#}$$

and, conversely

$$\gamma(\alpha(lfp\ F)) \subseteq \gamma(lfp\ F^{\#})$$

One of the main concerns of abstract interpretation is therefore to provide methods and techniques for the effective calculation of fix points [94, 3, 95].

In Chapter 7, we use the conceptual framework introduced here in order to develop a general purpose abstract interpreter for PHP based on our formal semantics for the language, discussed in Chapter 5.

4.6 Related techniques: symbolic and concolic execution

Techniques based on*Symbolic Execution* [96] are able to precisely analyse all program paths by treating program values as "symbols" and mimicking the execution of language statements on those symbolic values. Consider the example

```
$y = // ... integer input from user ...
$y = 2 * $y;
if ($y == 12)
   echo "BUG"; // triggering bug
echo "OK";
```

in this case, the integer value obtained from the user (and then assigned to `$y`) would be represented and stored as a symbol $n \in \mathbb{Z}$. After the second assignment, `$y` would then contain the symbolic value $2n$ (as a result of the arithmetic operation). In order to determine which branches could be taken in the conditional statement, a *path condition* is built for each branch; in the example, the

generated path conditions express the fact that, in order for the true branch to be taken, it must be $2n = 12$, while in order for the false branch to be taken it must be $2n \neq 12$. In this case, both path conditions turn out to be *satisfiable*, meaning that both branches could be taken at runtime given the appropriate inputs - i.e. none of them is unreachable. In order to take this to the next level, notice that a call to a constraint solver could easily compute, in this case, the exact input for which the true branch (the one triggering the bug) is taken: in fact, $2n = 12 \Rightarrow n = 6$, which can easily be calculated by modern SAT/SMT solvers. Therefore, the symbolic execution tool not only has found that a bug exist in the program, but it has also provided an input vector (the only one in this case) that triggers it.

Consider another case:

```
$y = // ... integer input from user ...
if ($y >= 0)
        $y = -$y; // $y is now negative
if ($y > 0)
   echo "BUG"; // triggering bug
echo "OK";
```

In this case, before the second conditional in evaluated, the system knows that variable `$y` holds a negative value (i.e. $y = -n \wedge n >= 0$). However, the path condition of the true branch of the second conditional looks like $y = -n \wedge n >= 0 \wedge y > 0$. This time, a call to a constraint solver reveals the *unsatisfiability* of the last constraint and therefore the fact that the associated program path is infeasible, allowing the tool to avoid reporting a false alarm.

From those simple examples, it seems that in fact symbolic execution is able to provide the best of both worlds: it is exhaustive as static analysis but at the same time it is "as precise" as testing, in the sense that it is able to catch real bugs providing concrete evidence for them. The price to be paid in this case is that, since program paths are explored exhaustively and no form of state-merging (Section 4.4.1) is used, the state-space quickly become too large - this phenomenon, known as *combinatorial explosion*, makes it harder for this class of techniques to scale to realistically sized codebases.

Concolic execution [97] attempt to bring together *conc*rete and symb*olic* execution. Consider the example

```
$x = // integer user input
$y = // integer user input
$z = 2 * $y;
if ($x == 100000) {
        if ($x < $z) {
                echo "BUG"; //triggering bug
        }
}
```

In this case, a bug is triggered whenever the value assigned to `$x` is `100000` and, at the same time, the constraint `$x < $z` holds. Notice that, intuitively, this concrete situation is fairly "rare", so that random testing would require a large number of attempts before being able to eventually find the bug. One possible solution would be to start by executing the program on a random test vector. Then, re-execute the same program *symbolically* on the same input but, this time, instead of following the exact same path, negate the very last path condition and, with the help of a constraint solver, obtain a new test vector which causes the program to take the opposite branch. This makes possible to "smartly" generate test cases achieving high path coverage. Concolic execution may suffer from the same limitations of symbolic execution, on which it is based. Furthermore, special care must be taken when programs exhibit non-deterministic behaviours, as re-taking the same path may produce different results. In general, symbolic execution based techniques show their weaknesses when, for example, dealing with higher-order data structures such as objects or when analysing loops depending on symbolic values. Chapter 6 describes our use of symbolic execution (paired with LTL model checking) in order to develop a simple verification tool for PHP scripts; known limitations of the approach are also discussed in the same chapter.

4.7 Static analysis tools for PHP

Analysis of PHP (and other web languages) is an important topic, given the prevalence of security flaws such as XSS, CSRF and SQL injection. There are many research and commercial tools that statically analyse PHP code, including Pixy [76], WebSSARI [77], PHP-Sat [98] and HP Fortify [99]. According to the respective papers or documentation, all of these tools have specific weaknesses related to language features that are hard to understand and analyse. For example, Pixy and WebSSARI do not follow taint flows across objects [76, 77]. In this Section we provide a brief survey of the existing static analysis tools and techniques which target PHP.

The problem with commercial tools is that it is usually difficult to obtain technical information about them in order to assess their inner functioning and the techniques they use. Furthermore, we noticed some tools are being advertised as static analysis tools while they actually perform very simple, mostly syntax-directed checks. We omitted this works from our brief survey, focusing only on tools for which substantial scientific literature was made available to the community.

Seminal work: string analysis.

One of the earliest static analysis tools explicitly targeting PHP is Minamide's String Analyser [100], presented a decade ago and based on the simple observation that the final output of a PHP program is a string (which the browser interprets as HTML).

Minamide's tool over-approximates the output of a PHP program as a context free grammar which is then compared against other grammars (using standard context-free grammar techniques) in order to check for malformed HTML, XSS vulnerabilities and more.

For example, given a program P and the context-free grammar $\mathcal{L}$ that over approximates its output, well-formedness of the generated HTML can be decided by deciding

$$\mathcal{L} \subset \mathcal{L}_{HTML}$$

where $\mathcal{L}_{HTML}$ is the context free grammar that specifies the HTML language [101].

While the problem of checking wether $\mathcal{L}_1 \subset \mathcal{L}_2$, for any two context-grammars $\mathcal{L}_1$ and $\mathcal{L}_2$ is

undecidable, the tool implements heuristics (based on common characteristics of HTML pages) in order to make the analysis more efficient.

Absence of XSS vulnerabilities can be checked by deciding wether $\mathcal{L}$ and the grammar described by the regular expression

```
* <script> *
```

are disjoint. This simple analysis is sound (it will find *all* XSS bugs) but very imprecise, as it rejects all programs that contain a script without considering that many PHP programs that output scripts are not malicious.

Minamide's work has been later specialised and extended by Su and Wasserman and applied to the detection of XSS [102] and SQLi vulnerabilities [91].

Data flow analysis and abstract interpretation. Pixy [76], developed by Jovanovic *et al.*, is an analysis tool for PHP which targets *taint-style vulnerabilities* such as XSS, SQLi and command injection. Instead of trying to compute an approximation of all the possible outputs as a grammar and then checking that it does not intersect the grammar of "dangerous outputs", Pixy uses standard data-flow techniques [93] in order to check that tainted data, i.e. data originating from the user, does not reach any *sensitive sink* (such as the `echo` statement) without having being previously sanitised (for example by an `HTMLCharacters` function).

Pixy's analysis is performed in three consecutive steps: an *alias analysis* which tries to compute, for each variable, the set of other variables that could be aliased to it, a *constant propagation* which tries to resolve `include` statements and finally a taint analysis itself, which utilises the data gathered by the previous steps in order to improve precision. One of Pixy's shortcomings is that it doesn't support objects [76]. However, object-oriented features appear to be popular among PHP programmers [103], and for this reason cannot be easily ignored by static analysis tools aiming at achieving extensive coverage of the language. Moreover, although Pixy pioneered the use of alias analysis for PHP, it deals with references only partially. In particular, only references between "simple variables", such as `$x =& $y` are considered, while references including e.g. array elements such as `$x = $y["foo"]` are not supported [76]. Also, significant imprecision in introduced when

performing assignments to array elements with non-literal keys, such as in `$x["foo"][$y]`. In such cases, the non-literal key is always treated pessimistically, i.e. by regarding it as "all possible indexed", and all possibly affected keys in the array are updated accordingly. While sound, this approach is rather imprecise.

WebSSARI [77] uses a type-system based approach in order to achieve the same goal of detecting taint-style vulnerabilities. In WebSSARI, a configuration file must specify, for every sensitive sink, a *precondition* about the arguments. For example, the precondition for the `echo` statement would require its argument to be untainted. Similarly, sanitisation functions have associated *post-conditions* specifying wether their output values have to be considered untainted. Data which possibly originates from the user is considered tainted.

One interesting feature of WebSSARI is the use of a *type-aware* taint lattice. A type-aware taint lattice improves the analysis precision by keeping track of the type of variables as well as of their taint status. The extra information makes it possible, for example. to avoid issuing an alarm when an integer originating from the user is printed (since only string values are actually harmful). Similarly, *casting* operations may be considered sanitisers: casting a (tainted) `string` to `integer` has the side-effect of sanitising the value.

One notable shortcoming of WebSSARI is the fact of considering all variable variables as tainted. We also noticed that, in the original paper, features such as arrays and objects, as well as `foreach` loops are not even mentioned [77]. The original WebSSARI tool has been improved during the years and is now powering the core of the commercial, closed-source HP-Fortify system [99].

In 2006 Xie and Aiken develop a tool using a "three tier architecture based on abstract interpretation" [77]. The idea behind their approach is to avoid analysing the program as a monolithic block, but instead analysing it iteratively at different levels of granularity, using the results of the previous step as inputs to the following, until the final analysis result is computed. The promise of this approach is to provide a more efficient and scalable analysis. This is accomplished by first analysing basic-blocks; the information about each building block that belong to a function is then composed and summarised to form *function summaries*. Function summaries are later composed

in order to produce the final analysis results.

Xie and Aiken's approach differs from the work discussed so far in that it it does not attempt to prove the absence of errors, but only to find the biggest possible number of them. In other words, the tool produces an under-approximation of the possible execution paths instead as an over-approximation. However, introducing unsoundness might in some non critical cases be desirable, in that allows for complete removal of false positives.

We also noticed, in their paper, a clear misunderstanding of the fact that arrays in PHP are copied by value. Consider for example the following snippet of code from their paper [77]

```
$hash = $POST;
$key = 'userid';
$userid = $hash[$key];
```

In the discussion that follows, they claim that "*The program first creates an alias* `$hash` *to hash table* `$POST` *and then accesses the* `userid` *entry using that alias.*". However, it is well known now that arrays in PHP are copied by value and not by reference (unless the special reference assignment construct is used). In the example above instead, the *entire* content of `$POST` would be copied into the target `$hash`.

Recent tools.

PHP-Sat [98] can perform different kind of checks such as finding common bug patterns and programming errors. The tool is not based on any semantic modelling of PHP, instead it uses simple syntax-directed checks. As far as we know, the tool has probably been discontinued, as we haven't been able to find updated versions online.

RIPS [104] combines approaches from previous work to deliver an efficient analysis that can find a large number of vulnerability with low false positive rate. It is based on three main conceptual steps. First, it builds a control flow graph (CFG) from the initial AST (as returned by PHP's parser). Second, it simulates each basic blocks and stores the results in *block summaries* as in [77], where each block summary maps the variables occurring in the basic block to *labels*. Labels contain information about data types as well as sanitisation or encoding *tags* among others. Arrays are

modelled by trees as in [76]. Finally, the tool tries to reconstruct each argument to a sensitive sink or user function by initiating a backward-directed string analysis [100]. If the argument turns out to be a constant, it is considered safe and the backward analysis is aborted. Otherwise, depending on the particular sink, the approximated argument (together with information about its encodings and sanitisation status) is passed to a specialised checker for the particular kind of vulnerability associated with the sink. The strength of RIPS lies in the effort put into engineering optimisations, which makes the analysis efficient, and in the precise modelling of a large number of sinks, encoding and sanitisation routines and vulnerability classes (around 900 in total), which makes it possible to drastically reduce the number of false positives. However, RIPS does not support recursive functions, and has only partial support for aliasing/references and object-oriented programming [104]. In particular, explicit assignment by reference (as in `$x = $y`) is not supported while they support parameter passing by reference). Since those features (among the most challenging of PHP) are handled optimistically, the tool is unsound.

RIPS has been extended by the same authors to support *second order vulnerabilities* such as *stored XSS* and *second order SQLi* [105]. While the underlying taint-analysis remain the same, the tool performs an additional step in order to identify *taintable persistent data stores* (PDS) such as databases, session keys and filenames. A PDS is taintable if user-defined data can possibly flow into it (such as a user comment being stored into a database). The idea is that when a flow from a PDS to a sensitive sink is identified (such as a user reading the comment on her browser), the tool checks wether the PDF in question was marked as *taintable* or not. If the PDS is taintable, the tool might perform further analysis and issue a vulnerability report. Otherwise, it defers the decision until the end of the analysis, when more data about the PDS in question might have been collected.

PHANTM [106] is a static analysis tool for reconstructing type information in PHP scripts and issuing warnings when possible type mismatches are detected. Consider the following example from [106]:

```
$conf["readmode"] = "r";
```

```
$conf["file"] = fopen($inputFile, $conf["readmode"]);
$content = fread($conf["file"]);
fclose($conf["file"]);
```

Since `fopen` returns either `false` (in case the file is not found) or a reference to a file (a `resource` in PHP terminology), there is a potential type error in line 3, since the value `false` may be passed to `fread` which instead expects a reference to a file. In this case, PHANTM reports a warning, which could be fixed e.g. by guarding the `fread` operation with a conditional such as

```
$conf["readmode"] = "r";
$conf["file"] = fopen($inputFile, $conf["readmode"]);
if($conf["file"]) {
        $content = fread($conf["file"]);
        fclose($conf["file"]);
}
```

After the fix, PHANTM correctly recognises that, this time, the argument to `fread` can only be of the correct type. This is achieved by using *union types* and *type refinement*, as follows. After the call to `fopen` in line 2, the type `false` $\cup$ `resource` is assigned to `$conf["file"]`. Inside the conditional branch, however, this type is refined as `resource`, since the value `false` cannot possibly evaluate to `true` and therefore it couldn't cause control to pass to the true-branch of the conditional.

According to the authors themselves, the tool produces a high number of false positives, and has a few important limitations in terms of the supported portion of the language. For example, references (e.g. `$x =& $y`) as well as variable variables and access to non-literal array/object fields (e.g. `$x[$y]`) are not supported. The use of expressions with side-effect inside conditional guards is also forbidden (in which case a "bad practice" error is issued), and *conditional declarations* of functions or classes are not considered (i.e. only the first declaration is processed, and the subsequent ones are ignored). Furthermore, in [106] there is no mention about other important language features such as recursion, exceptions or `break` and `continue`, which we believe are also non supported.

More recently, a static analysis framework for PHP has been proposed by Hauzar *et al.* [107]. The framework has the goal of being extensible (i.e. users can define their own analysis on top of it) and it is based on a two-phase architecture. The first phase essentially targets the more dynamic features of the language (variable functions and variables, non-static assignment targets etc.) and tries to resolve them so that the subsequent user-phase could be more easily defined without having to deal with the abstraction of those challenging features. While, in principle, both analyses could be user-defined, the authors' prototype include a constant-propagation analysis for the first step (using sets for strings and intervals for integers) and a taint-analysis for the second step, in order to target taint-style vulnerabilities such as XSS and SQLi. In particular, the first-phase analysis attempts to accumulate the possible values of variables so that control flow and heap information could be determined, then builds an intermediate representation (IR) on which the user analysis is performed. The intermediate representation encodes both evaluation order and data flow, and constitutes the language model on which they base the analysis. Consider for example the snippet

```
$x[$y] = 1;
```

The first-phase analysis attempts to accumulate the possible values for `$y`, either as a set of strings/integers or as an integer interval. Then, the assignment is modelled by updating the values of all possibly affected variables. Once this information is encoded in the intermediate representation, a user-analysis could be defined without having to explicitly consider the complex behaviour of assignment in case of unknown target. Instead, the analysis could be defined by just specifying what happens for a simple assignment, since the complex, dynamic behaviour is dealt with in advance by the framework. The framework supports many of the features that were not considered by previous work such as aliasing, exceptions, `include` and `eval`. We are not sure wether they support e.g. `break` and recursion, since this is not mentioned in their paper [107].

Although not directly targeting security vulnerabilities, the *include resolution* analysis proposed in [108] is worth mentioning. The analysis is inspired by the assumption, validated in an empirical study by the same authors [103], that in real-world PHP applications the vast majority of dynamic file includes can either be resolved to a single file or to a small set of them. The authors

provide two flavours of the analysis, one operating on a single file and one on *programs*, i.e. PHP scripts that can be run directly either on the web or as command line tools, with the latter being built on top of the former. Both the analysis are lightweight and have been evaluated on a large corpus of popular open source projects [108]. Essentially, for each `include` found in the program, the analysis attempt to reconstruct the argument by replacing known constants for their value and by simulating simple string operations. If the results is a literal string, then the included file is looked up following PHP's (complex) semantics for file inclusion. Otherwise, the incomplete string constructed by the algorithm is used to construct a regular expression which will be used to match a set possible files. This kind of analysis is relevant because dynamic include has been long considered one of the most challenging features of PHP from a static analysis perspectives and this work shows that while the problem can not be solved in principle, it can at least be managed. Pixy attempted to solve the same issue by performing a preliminary constant propagation [76], but it was less precise because it didn't take into account the actual semantics of file inclusion in PHP (i.e. it just tried to reconstruct strings by using constant propagation).

Conclusion and perspective.

From our survey, it emerged that existing static analysis tools for PHP generally fail to model the challenging features of the language such as aliasing and object orientation. Non-supported features are generally treated optimistically (their usage is always considered safe) introducing unsoundness (bugs might be missed) and making tools unable to provide strong guarantees in critical industrial settings. Alternatively, such features may be treated pessimistically (their usage is always considered unsafe) but this has the potential of causing a large number of false positives to be introduced. We also noticed misunderstanding and confusion about the semantics of the supported features themselves, notably arrays and the way they are copied and referenced (as e.g. in Xie and Aiken [77], explained above). We believe that the problems above have their root in a poor understanding of the language itself and can therefore be addressed by providing tool designers a formal, unambiguous mathematical specification of the language to use as a reference. Having such a specification as a starting point for tool design can provide guidance while perhaps

also demystifying what are commonly believed to be "hard features". For example we noticed that object orientation is often not supported in static analysis tools because it's considered a challenging feature. Our semantics completely define objects (and operations of them) in terms of arrays, making it easier to support them in a verification tool. In fact, modulo the additional challenges due to the visibility (i.e. `public`, `protected`, `private`) of their fields, most of the techniques we have developed for arrays can be applied to objects. Moreover, having a mechanised mathematical specification of the language paves the way to the development of *certified* analysis tools which can provide strong guarantees about the software being analysed, for example by certifying the absence of a certain type of bugs. This havs the potential to create a new market of highly-trusted, certified verification tools to be used in highly-sensitive industrial contexts.

In Chapters 6 and 7 we present our own contributions - a simple program verifier based on LTL model checking, and a general purpose static analyser based on abstract interpretation, based on $\mathbb{KPHP}$, our executable formal semantic for PHP introduced in Chapter 5. Being based on a trusted semantics which faithfully models the full core language, our tools are able to deal with aliasing, objects and other features of PHP which are sometimes not properly handled in related work.

Part III

Contributions

Chapter 5

$\mathbb{KPHP}$: an executable formal semantics of PHP

In this Chapter, we describe $\mathbb{KPHP}$, our formalisation of the operational semantics of PHP using the $\mathbb{K}$ framework. $\mathbb{KPHP}$ is the first formal (and executable) semantics of PHP to date. The semantics formally encodes our understanding of the language (including its darkest corners) and provides an ideal starting point for the development of a range of tools for the language, as demonstrated by our two initial case studies - a verification tool based on LTL model checking (Chapter 6) and a general purpose abstract interpreter (Chapter 7).

As discussed in Section 3.2, at the time we started our formalisation there was no official document providing a specification of PHP. On the other hand, the recent PHP specification introduced by Facebook [75] still provides a very abstract memory model and does not specify the more involved language features and corner cases, as discussed in Section 3.2. Hence, the development of our semantics was largely test-driven. The choice of specifying the semantics in $\mathbb{K}$ meant that at each stage of this work we had a working interpreter corresponding to the fragment of PHP we specified up to that point. This made it possible to test critical semantics rules as they were being developed. For this ongoing testing we wrote snippets of PHP, and compared the results from our interpreter with the ones from the Zend Engine, which is the *de facto* reference

interpreter.

Other reasons for choosing to develop the semantics in K are its user-friendliness, the intuitive way in which programming language concepts are described and the several helper tools offered by the framework (such as debugging, state-space exploration, model checking). A digression about the author's personal experience may be helpful to perspective students or researchers who are considering to get involved in similar efforts: the framework of choice for our semantics, K, was initially discovered "by accident" while performing a literature review about existing tools and frameworks for semantics engineering (Section 2.2.1); after having read for the first time about the tool, we decided to give it a try, mostly out of curiosity, and started working through the online tutorials, eventually ending up with an executable formal semantics for a toy imperative language (similar to IMP, discussed in Section 2.2.2). Eventually, after a couple of weeks spent experimenting with the framework and extending the language with real-world PHP features, we somehow knew that this particular framework was the right choice for our task, so we decided to use it in the development of our formal semantics of PHP. The lesson to be learned is to *experiment* with different tools, read about them and stay up-to-date. Tool-supported semantics engineering is an active field and new and better tools are being developed (while existing ones are being improved); we believe it is important to research and experiment with different tools in order to find the one that best suits the given task with its requirements and constraints.

Now that we have mentioned the strengths of our framework, it is probably a good time to also discuss its main weakness: the general "slowness" of the generated interpreter. As an example, consider a standard factorial function (defined recursively)

```
function factorial($n) {
        if ($n == 0)
                return 1;
        else
                return $n * factorial($n - 1);
}
```

and a program that calls it to compute and display the first 10 factorials

```
for ($i = 1; $i <= 10 ; $i++)
        echo factorial($i) . "\n";
```

producing the output

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

Using the Unix `time` command, we measured that the Zend PHP interpreter takes $0.098s$ to run it while $\mathbb{KPHP}$ takes $11.379s$; simpler, non recursive programs usually take about $1s$ to run, which we believe is mainly spent on the initialisation of $\mathbb{K}$'s underlying machinery (i.e. at every invocation, $\mathbb{K}$ starts a JVM, opens Maude and initialises several other helper tools). This phenomena is in fact expected and common to other semantics frameworks (see Section 2.2.1) and we do not consider it as a limitation of our work: the main focus on this contribution is the introduction of the first formal semantics of PHP, and our $\mathbb{K}$ formalisation provides a specification of our ideas; the fact that $\mathbb{K}$ is also executable provides us with a proof-of-concept preliminary interpreter for free (which allows us to test the semantics), however this is not to be considered the final implementation. In fact, in future work, we plan to use the existing $\mathbb{K}$ specification as a starting point for the development of an industry strength PHP interpreter and static analyser, using a general purpose (and faster) programming language.

Our semantics of PHP is trusted, as we extensively tested against the Zend Test suite [78], the conformance test suite distributed with the Zend engine and used to benchmark PHP interpreters and tools (including Facebook's HHVM [72]). This ensures that the generated interpreter behaves in the same way as Zend, but notice that this fact alone does not provide any correctness guarantee:

what happens, for example, when running test cases which does not belong to the Zend test suite? This depends on several factors, including the *quality* of the test suite (e.g., how comprehensive is it? How passing a test generalises to similar situations?) but also the extent in which the development has been further validated by running something other then the test suite itself. For example, after passing all tests, one should also try new code (such as real-world code or a different test-suite) and check that there haven't been any *overfitting* and that the expected behaviour generalises properly. Discussing testing and its issues in full generality is clearly our of the scope of our Thesis; we discuss how we tested and validated KPHP in Section 5.4.

Our semantics is vast (more that 1200 rules), so we only provide a roadmap through it, selectively explaining the most important features, and referring the reader to `http://phpsemantics.org` for full details. Although we support most of the core language, we have not (yet) implemented a number of non-core features, and in particular: bit-wise operators, most escape characters, regular expressions, namespaces, interfaces, abstract classes, iterators, magic methods and closures; we discuss limitations and future work in Section 5.5.

During the discussion of the semantics, we also look more in depth, to uncover difficult "corners" of PHP. While some of the examples are well-known PHP issues, the others are our original observations, discovered while developing the relevant semantics rules. Although some PHP experts may be aware of these cases, they are not part of the mainstream knowledge about PHP, and are hence worth discussing. Moreover, we show how most of those widely regarded "odd" and commonly misunderstood behaviours are in fact clearly described by our semantics.

5.1 Syntax

For parsing, we use the PHP grammar from `PHP-front`, a package for generating, analysing and transforming PHP code used by the `PHP-sat` [98] project. The grammar is in the *SDF* [109] format, which can be passed to the `sdf-2-kast` tool [110], which generates an executable parser whose output is in the KAST format, the standard AST encoding used by the K tools [35]. To provide

a feel for the format, consider the following simple code

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

The corresponding AST produced by our parser looks like the following

```
'Expr(
      'Assign(
        'Variable('Simple("x")),,
        'Variable('Simple("y"))))
```

In particular, notice that all AST nodes are identified by the ' character and that arguments are separated by double commas. We mention right now that our parser has a few limitations. Notably, it does not provide line-number information, although we have provided a workaround (see Chapter 7). We discuss further limitations in Section 5.5.

5.2 Semantics

5.2.1 Memory Layout

Values

PHP supports three categories of data types: *scalar*, *compound* and *special*. Scalar types are the `boolean`, `integer`, `float` and `string` types, compound types are the `array` and `object` types, and special types are the `NULL` and `resource` types. The `resource` type is used for files and other external resources, and is left for future work.

Scalar	Compound	Special
`boolean`	`array`	`resource`
`integer`	`object`	`NULL`
`float`		
`string`		

Table 5.1: PHP data types

For simplicity, we model scalar types by the corresponding built-in types of K, although there may be some subtle differences for example in the approximations made by floating-point computations. For example,a statement such as `echo(10/3)` prints `3.3333333333333` in Zend and `3.3333333333333335` in KPHP. Test cases explicitly checking this kind of functionality will fail and therefore will need to be manually reviewed [1]. In the future, we plan to enhance out test harness so that it automatically detects and deals with this kind of false positives. We currently do not plan to implement/model floating point arithmetic ourselves, as we consider this task to be squarely out of the scope of this project (which instead should be to focus on the features specific to PHP).

Arrays and objects

We model arrays as pairs `array(C,EL)` where `EL` is a list of array elements and `C` is an optional *current element*. Array elements are represented as triples `[k, v, l]` where `k` is an `integer` or `string` *key*, `l` is the memory location where the actual value is stored, and `v` a *visibility* attribute (discussed below).

Objects are triples `OID(L,CL,ID)` where `L` is the location of an array containing the fields of the object, `CL` is the name of the object's class and `ID` is a unique numeric object identifier (usually an implementation-dependent instance counter).

The visibility attribute is always `public` for *proper* array elements, whereas it can be also `protected` or `private` for objects fields. As shown in the object inheritance example of section 3.2.2, the correct handling of visibility of object properties is subtle. In particular, array elements are identified by a combination of the key and the visibility attribute. In that example, the field array of `$obj` contains two separate entries `["id",public,l1]` and `["id",private(par),l2]`, necessary to resolve the right element depending on the context.

Finally, classes are 4-tuples `Class(SC,IV,MT,SV)` where `SC` is the name of the superclass, `IV` is the list of instance and static variables, `MT` is the method table, and `SV` is a pointer to the scope

[1] Fortunately, we only found one such case in the Zend test suite

holding static variables (those shared across all objects from the given class).

Arrays play an important role in $\mathbb{KPHP}$ since they not only are a data-type in their own right, but are also used as building blocks for objects, and - as we will see briefly - scopes. The modelling of arrays, especially when considering the interaction with other features such as aliasing, was one of the main challenges of this and related efforts (e.g. [76, 77]).

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

The reader might wonder *why* we modelled things in this particular way, and why we made non-obvious design decisions such as equipping array fields with a `public` visibility attribute. This last choice is easily explained: during our study of the language, we realised that objects exhibit the same functionality as arrays, modulo the visibility issues that arise when taking classes into account (i.e. array elements can always be accessed, while object properties can be accessed only when visible). In this setting, modelling arrays and objects independently would require the introduction of a significant amount of semi-duplicated code, resulting in an error prone and inelegant result. For this reason, we decided to unify the treatment of arrays and objects by representing them with a single shared data structure and by designing the core rules of our semantics so that they operate on both, resulting in a shorter, elegant, unified model. Since, in this general model, both arrays and objects share the same underlying data structure, array fields ended up being equipped with a default `public` visibility, while object properties may also have `protected` or `private` visibility; this supports our intuition that, while array elements can always be accessed (i.e. they are public), objects properties can only be accessed when certain (visibility) conditions are met. In general, since no specification of PHP was available to us during the development of our work, we took some freedom in our decision-making process, focusing on two goals: modelling the language behaviour correctly and producing a clear, elegant and easy to understand model.

ZVals

Following the online PHP documentation, in the memory we wrap values into four-tuples `zval(Value, Type, RefCount, Is_ref)` [82]. Each `zval` contains a value, its type, a reference counter keeping track of the number of variables currently pointing to the value, and a boolean flag indicating whether or not the value is aliased. In Zend, the flag `is_ref` is used for implementing the array *copy-on-write* optimisation [62, 82]. We include it just for completeness, since in our semantics, `is_ref` is true iff `refcount`>1. We plan to model the copy-on-write semantics in future work (perhaps as a module which can be optionally imported in the semantics). We discuss copy-on-write and the issues it may cause in 5.2.3.

We define a number of internal low-level operations which manipulate zvals (`zvalRead`, `incRefCount`, etc.), and use them as building blocks for defining higher level functions (`read`, `write`, etc.), providing the illusion of operating directly on simple values, increasing the modularity of the semantics. For example, in order to write a value `v` into a memory location `l`, we use the macro `writeValueToLoc(l,v)` which creates an empty `zval` at location `l` and initialises it with the given value `v`, its type and reference counting information. Similarly, when performing assignments (in particular by reference), we let macros take care of updating the `RefCount` and `Is_ref` fields as needed. All this will become clearer in the following.

Memory

The heap, which is contained in the `heap` cell, is a map

$$\mathbb{H} : loc \rightarrow zval$$

where loc is a countable set of locations $l_1, \ldots, l_n$. Fig. 5.1 shows the heap after executing the program

```
// $x is initialised. It's "current" element is the first, "foo" => 5
$x = array("foo" => 5, "bar" => 5);
$y = 5;
```

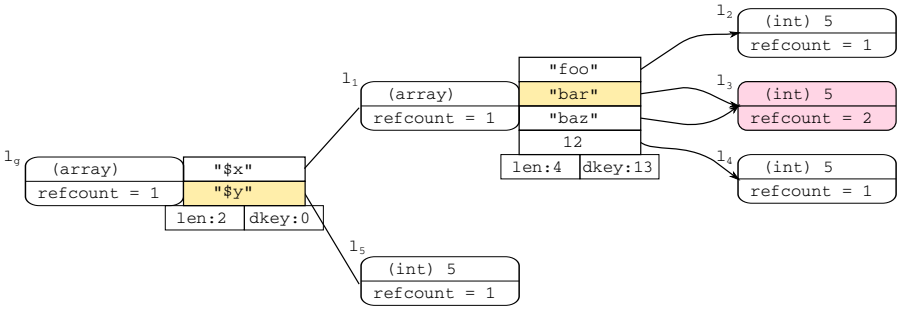


Figure 5.1: Example heap, where the reserved location l_g contains the global scope.

```
// Advance $x's current element by one position.
next($x);
$x["baz"] = &$x["bar"];
$x[12] = 5;
```

where the elements pointed to by the array *current* pointers are shaded (in yellow), and shared `zval`s are shaded (in red), and l_g is the location containing the global scope, which is a special array where both `$x` and `$y` are defined. We have shown in Section 3.2.2 how the global scope can be accessed directly via the variable `$GLOBALS`. In fact, just like in JavaScript, it is convenient to represent all PHP scopes as heap-allocated arrays [20]. The location of the array designated as the current scope is always available at runtime so that other rules can access it in order to perform their tasks (such as lookup or assignment). To further illustrate how the memory in $\mathbb{KPHP}$ looks like, Figure 5.2 shows the heap after the execution of `$x = new A()`, where `A` is a class defining the properties `data` and `counter` (assuming default values of `5` and `"hi"` respectively). The example shows similarities and differences between how arrays and objects are represented. Objects are essentially defined by adding a layer of indirection on top of "standard" arrays, and when it comes to the manipulation of their properties (reading and updating), we are essentially relying on the existing semantics of arrays. However, because the `OID` is the actual value being returned and stored when a new object is created, it follows that objects, unlike arrays, are always copied and passed around by reference. A copy by value can be performed via the builtin function `clone` which essentially performs the same copy strategy that happens normally when assigning arrays.

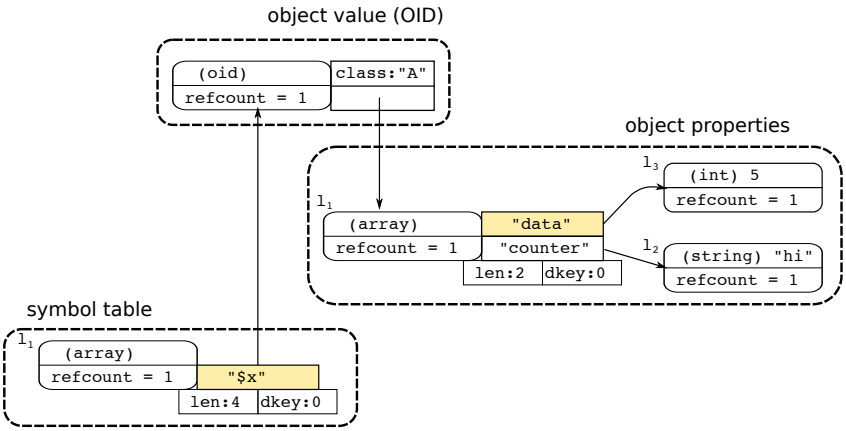


Figure 5.2: Building objects as arrays

References

Several programming languages internally use references, and so does PHP, but with its own original twist. Consider running a simple program `unset($y)` on the state shown in Fig. 5.1, where the command `unset`, according to the online documentation, "destroys the specified variables". If the argument `$y` were evaluated to a value, we would reach `unset(5)` which is nonsensical. Even evaluating `$y` to the location l_5 would not be the right choice, since in that case we could successfully free the location, but not remove the link from `$y` to l_5 (which is stored in the array at l_g). This is just an example of a general class of cases, which imply that variables need to be evaluated to *references* of the form `ref(L, K)` where `L` is the address of the array or object containing the variable, and `K` is the variable name. When the actual value stored in the variable is needed, further steps of reduction can be taken to resolve the reference. This is not a trivial process, as the lookup depends on whether the reference appears on the left or right hand side of an assignment. Consider the code

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

where neither `$x` nor `$y` have been initialised. The first step is to evaluate the variables obtaining the references `ref(`l_g`,"x")` and `ref(`l_g`,"y")`. On the left-hand side, since `"x"` is not an entry of the (array) scope l_g, it will be created, adding a link to a fresh location. For the right-hand side, since `"y"` is also not present in l_g, `NULL` will be returned and written to the fresh location.

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

```
function mod_x() {
        global $x;
        $x = array('a','b');
        return 0;
}
$x = array(1, 2);
$x[0] = mod_x();
var_dump($x);
```

which outputs

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

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

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

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

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

We handle this via the `get_l` and `get_r` internal operations, which perform two different variations of reference resolving. `get_r`, generally invoked on the right-hand-side of an assignment,

recursively resolves the reference starting from the current scope until it gets to a location (which is returned). If an undefined variable is encountered, `get_r` returns the special location `locNull` which causes the lookup to be aborted and the value `NULL` to be returned. `get_l`, generally invoked on the left-hand-side of an assignment, is similar to `get_r` except for the fact that when it encounters an undefined variable it allocates it on-the-fly into a fresh location and returns that location so that the lookup can succeed.

The reader might have noted that in order for `get_l` to be able to initialise variables on-the-fly, finding an undefined "link" in the path encoded by an `lref` is not enough: it also needs to know the expected type of it. In the example above, if the left-hand-side of the assignment were simply evaluated to `lref(lref(ref(l_g, y),0),x)`, then there would be no way to know, when resolving the reference, that `$y` must be initialised as an array, and that its property with index `0` would be initialised as an object. Furthermore, the property `x` of the object would also have to be initialised to some value. We use the value `NULL` for this, but any initialisation will do since this value will generally be re-assigned (to 42 in the example).

To allow this, we encode the type information in the `lref`s themselves so that `get_l` and `get_r` could make use of it during the process. Hence, the left-hand-side of our example evaluates to the `lref(lref(ref(l_g, y),0,array),x,object)` which, when resolved via `get_l` results in the initialisation of `$y` as indicated by the source code.

The reference mechanism of KPHP is quite involved but we believe it provides an elegant unified solution to the problem of resolving references involving simple variables as well as arrays or objects, in all possible contexts.

5.2.2 Configuration

The configuration of KPHP, which represents the global state of the abstract machine, consists of 42 cells, and is shown in Figure 5.3. The `script` cell contains details of the script being executed, and in particular a cell `k` with the actual program in the meta-variable $\$PGM$. The `declarations` cell contain declarations hoisted in an initial parsing phase (detailed in 5.2.4). This is necessary

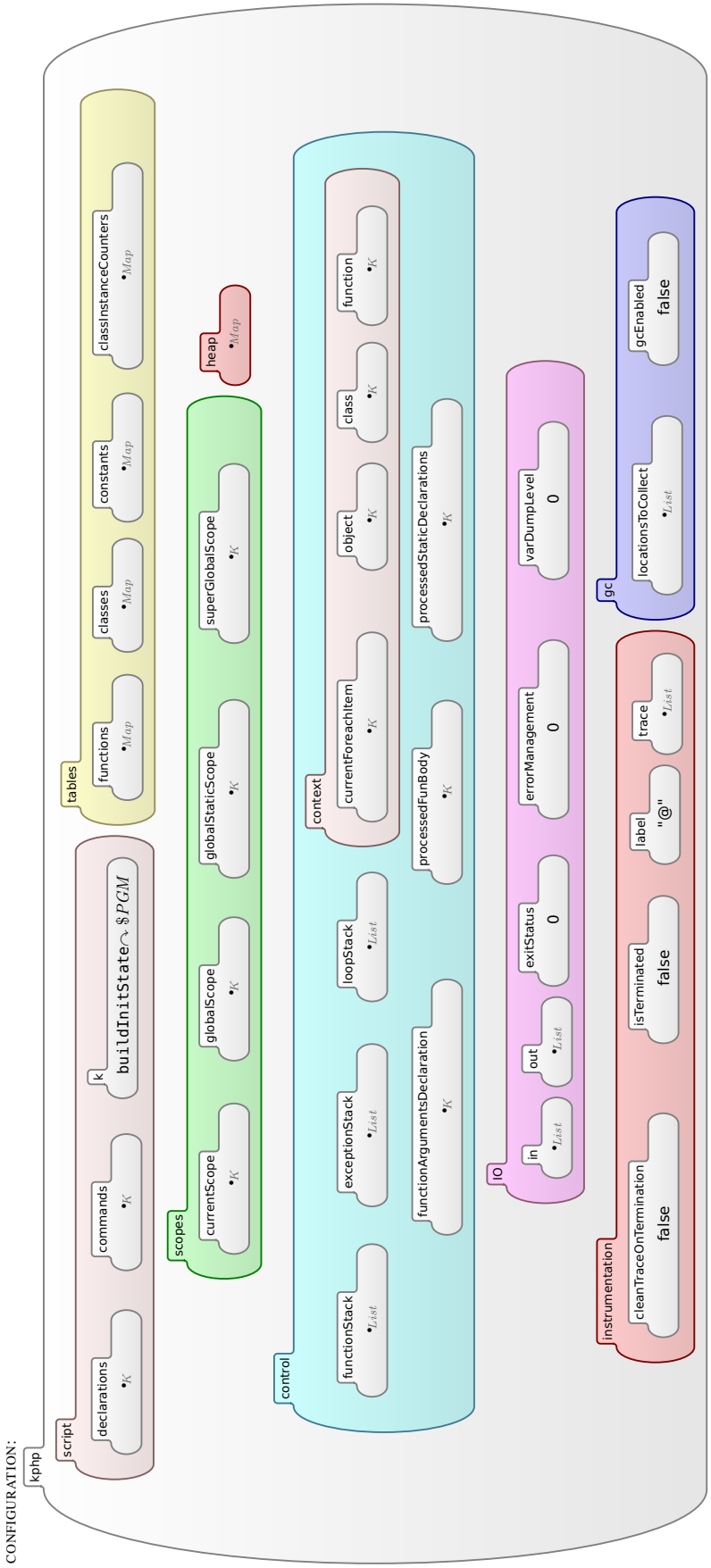


Figure 5.3: Configuration for $\mathbb{KPHP}$.

because a function or class defined in a PHP script can be used by code appearing before the actual declaration. During the initial phase, the program is split into two parts: commands (stored in the `command` cell) and declarations. Once the separation is complete, the program is reassembled by putting all function and class declarations *before* the commands, with the effect of ensuring that declarations are available at each program point, even before their actual declaration. The reassembled program is then moved inside the `k` cell for execution, which proceeds as normal. Notice, however, that the program to be executed is preceded by the auxiliary operation `buildInitstate`, which takes care of initialising the environment with a set of commonly used build-in functions and classes (such as the `Exception` class and functions like `var_dump`). The `tables` cell contains function, class and constant definitions, plus a map from class names to integers, keeping track of the number of instances that were created for each class. The `scopes` cell contains pointers to the various (global, super-global, current) scopes in the heap, including a separated scope for static variables declared outside of any function. The `control` cell contains the function, exception and loop stack, and information about the current object and class. The *function stack* is particularly important in that allows functions to support recursion; its concrete usage will be clarified later in this chapter. Finally, the `control` cell also contains a number of auxiliary cells (such as `functionArgumentsDeclarations`) which we use to store bookkeeping information generated and processed during function calls. We clarify the purpose of those cells when appropriate in the context of our later discussion of the semantics. The `IO` cell contains the input and output buffers, which K automatically connects to `stdin` and `stdout` (detailed later in 5.2.4), as well as the `exitStatus` cell, representing the final exit status of a script (e.g. `0` for correct termination, `1` for error etc.). The `errorManagement` cell contains an auxiliary error code, for testing purposes; our semantics is instrumented so that, in case of failure, the `errorManagement` cell is updated accordingly. For example, when a construct that we do not support is encountered, the error code is set to `1`. Similarly, if the code triggers any warning, the code is set to `2`, and so on. Our test harness then inspect those code in the final result and organises the HTML report accordgly.

The `instrumentation` cell gathers meta-information for analysis purpose, such as the trace

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

5.2.3 Copy-on-write in the Zend engine

Copy-on-write is an optimisation mechanism used by the Zend engine in order to increase performance and reduce memory usage [111, 112]. It essentially consists in deferring the actual copy of variables until the point where either the origin or the destination variable is actually updated. Suppose `$x` is a variable containing a large data structure. Suppose a program contains the assignment `$y = $x`, but then never actually writes on `$y` (note that this also applies the case when `$x` is passed to a function). The idea is that it is clearly not necessary to *copy* the whole data structure unless the assigned copy needs to be updated.

Instead, when using copy-on-write, the assignment is actually performed by reference. However, the `is_ref` field (see 5.2.1) of the involved ZVal is *not* set to true, even if after the assignment there are 2 variables actually pointing to it and its references counter is set to 2. Only when either `$x` or `$y` are written the actual copy is performed. In this case, the engine looks at the value of the `is_ref` field, and since it contains the value `false`, then the aliasing is known to be only temporary, and therefore the actual copy is performed (this operation is known as "splitting" in copy-on-write terminology). Hence, the copy is performed *on write* instead as *on assignment*. On the other hand, if the assignment was initially performed by reference, i.e. using $y = \&x$, then the `is_ref` flag would be set to `true`, encoding the fact that no actual copy is necessary. Suppose instead that the variable `$x` was already (explicitly) aliased before the assignment as in the following program

```
$z =& $x;
$y = $x;
```

In this case, the assignment `$y = $x` immediately forces the copy because, otherwise, there would be an aliasing between the variables `$x`, `$y` and `$z` and the engine would lose track of the fact that, while the aliasing between `$x` and `$z` was introduced on purpose, the one between `$y` and `$x` was only temporary and due to the copy-on-write strategy.

An optimisation such as copy-on-write is supposed to be *transparent* - the programmer does not need to know the mechanism is in place. In other words, the semantics of PHP with copy on write and the semantics of PHP with copy on assignment should be equivalent; in other words, running the same program with or without copy on write, should always lead to the same results, no matter what input is provided.

In 2009, Tozawa et al. [62] exposed (and reported to Zend) a flaw in the copy-on-write implementation used by the Zend engine, so we were quite optimistic in the fact that today copy-on-write was in fact correctly implemented, and that its presence could not be detected by running programs. Unfortunately, during our testing we found some erratic behaviours which we believe are due to the presence of copy-on-write (discussed in 5.2.4). Some of those behaviours have been discussed online [113] and reported as bugs but is seems that no conclusion has been reached by the PHP developers community. Despite this, we still believe that an optimisation such as copy on write should be implemented correctly (or avoided altogether), and that the bizarre behaviour we found should be properly addressed by the language developers, instead as being formalised by us. Therefore, in KPHP, we do not model copy-on-write and instead we rely on the simpler copy-on-assignment strategy. This choice was dictated by the fact that we want to keep the semantics as simple as possible and be focused on the meaning on the language itself, not on the particular engineering optimisations that the Zend engine (or another implementor) may use. In the following, while adopting a copy-on-assignment strategy, we explicitly discuss cases where copy-on-write is exposed.

5.2.4 Semantic rules

Overview

Each non-trivial language construct is described by several rewrite rules, each performing a step towards the full evaluation of the construct. Conceptually, the evaluation process happens in three steps. First, *structural* and *context* rules are applied. Their role is to rearrange the current program so that other rules can be applied. These include for example *heating* and *cooling* rules,

which move the arguments of an expression to the top of the computation stack (cell `k`) and plug the results back once evaluated, and *desugaring* rules. Next, *intermediate* rules apply. Their role is mostly to pre-process arguments. For example, they convert types, resolve references, or read from memory. Finally, *step* rules apply. They give semantics to the actual language constructs, and cause the term being evaluated to be consumed, returning a value where necessary, so that the computation may progress. See 2.2.2 for details on $\mathbb{K}$ rules.

Besides rewrite rules, $\mathbb{K}$ definitions also include *functions*, which do not have side effects on the configuration. In our semantics, we mostly use functions to define logical predicates to be used as side-conditions in other rewrite rules. For example, the following rule takes a value as argument and determines wether it is a numeric value (i.e. an `int` or `float`) or not.

```
syntax Bool ::= Value "isNumeric" [function]
rule V isNumeric => isIntType(V) orBool isFloatType(V)
```

where `isIntType` and `isFloatType` are also functions defined via standard pattern matching of the argument. Since the rule is declared as a `function` of sort `bool`, it can be used as side condition in other rules:

```
// 1st case
rule toyRule(V:Value) => "I'm a number"
when V isNumeric
// 2nd case
rule toyRule(V:Value) => "I'm something else"
when notBool(V isNumeric)
```

Overall, the semantics comprises over 1,200 definitions: more than 700 are proper transition rules, the others are auxiliary definitions. In the following we show and discuss a few selected examples in great details. The complete up-to-date semantics can be found at `www.phpsemantics.com`.

Program transformation

In a few cases, we perform some form of program transformation, mainly for making it possible to refer to functions or classes before their declaration and for initialising the static variables of a function. We now discuss both cases.

Hoisting. In PHP, functions and classes can be used before their definition. For example the program

```
sayHello();
function sayHello () { echo "hello world" }
```

is a valid program which prints the string `"hello world"`.

In order to allow this functionality we perform a preliminary refactoring of the program before executing it. In particular, we traverse the whole top-level nodes of the AST and we examine each one. If the node is a function or class declaration, we remove it from the original AST and we add it to a second AST, in the same order. At the end of the visit, there are two lists of AST nodes: one, say D, containing all the program declarations and one, say S, containing the statements. We then put the concatenation $D :: S$ into the K cell for execution.

This has the effect of processing declaration first so that they may be used in any point in the program even if it happen to be before the actual point in which the function or class is first declared (as in the example above). Note that it is not required to perform this operation on functions or classes defined inside other functions or classes. Instead, in such cases the function/class becomes available only if the enclosing function is called (this mechanism, expected in PHP, is called "conditional definition" [114]). This helps keep our mechanism simple, as it doesn't have to recursively visit every top-level node encountered in the initial visit. We perform this code transformation once and for all before the program is executed and therefore it can be thought of as a *compile-time* operation.

The mechanism is implemented using a standard *visitor pattern* which is built-in in K. We omit showing the rules here, but the interested reader can read the full rules in the file `program-transformation.k` of the semantics available at `http://phpsemantics.org`.

Static variables. In addition to the local function variables (including parameters), PHP allows to declare static function variables. Static variables are persistent, i.e. their value is preserved across different invocation of the same function. Consider the following example from the online documentation

```
function test()
{
    static $a = 0;
    echo $a;
    $a++;
}
```

The first time `test` is called, it will print `0`. If called again it will print `1`, `2` and so on.

$\mathbb{K}$PHP supports static function variables. In order to do that, we found it helpful to perform a preliminary transformation of the function body. The preprocessing is done once and for all and it changes the function body - every time the function is called, the modified function body is the one that is executed.

In order to explain the mechanism, we shall first discuss how we represent static variables. As discussed in detail later in this section, when a function is declared, a new entry `f(...)`, namely a constructor `f` containing various pieces of information (such as the function body, the parameters etc.), is created and stored in the `<functions>` cell. Among the information stored into `f` is a pointer L to an array containing the static variables. The idea is to store static variables into this scope (which is persistent, and whose address is stored into the function definition itself) so that they are preserved across different calls, while standard local variables are stored into the function's local scope (which is created anew for each function invocation and destroyed when the function returns).

We implement this behaviour as follows. When a function is declared a preprocessing step is taken in order to initialise the function's static scope with all the static variables declared in the function body with their initial value (which is NULL by default in case the static variable was declared without providing an initial value). This is done by traversing the AST (using $\mathbb{K}$'s visitor

pattern) and performing, for each declared static variable, a standard assignment of its initial value to the variable itself, in the context of the function's static scope. At the end of this process, the (initially empty) scope is populated. Static declarations, once the corresponding assignment has been generated and executed, are replaced with a special instruction (only available in the semantics but not to the developer) which, when executed at runtime, assigns by reference the static variable (which, by that time should be allocated into the static scope) to an homonymous variable in the function's local scope. This causes static variables to be visible alongside local variables and function parameters and doesn't require different lookup mechanisms: all variables, static or otherwise, are visible and accessible in the function's local scope. On the other hand, when the function returns, its local scope is destroyed, while its static scope is not. Indeed, its location will remain in the function's entry in the `<function>` cell so that static variables and their values could be maintained and be available at the next call. Function call and return mechanisms are discussed in great detail later in this section.

Assignment

The rules for assignment are reported in Figure 5.4. Assignment is a binary expression whose arguments are evaluated left-to-right. We model this by two `CONTEXT` rules that enforce that evaluation order: rule (A) does not prescribe a type for the wildcard `_` variable, that can match any term, including an unevaluated expression, resulting in the left-hand-side being evaluated first. Instead, rule (B) can apply only after the first argument is evaluated to a `KResult`, causing the right-hand-side to be evaluated last. *Context rules* are a generalisation of the `[strict]` tag (see 2.2.2) and cause the K tool to automatically generate heating and cooling rules that realise the specified evaluation strategy, where the subterm to be evaluated is indicated by the `HOLE` special K variable. In an assignment, the LHS will evaluate to a reference. Rule (C) resolves the reference to a location by calling the auxiliary function `convertToLoc`. If the RHS is a value, rule (D) puts the internal operation `copyValueToLoc` at the front of the `k` cell, and puts the value `V` to be returned as a continuation. `copyValueToLoc` takes care of writing `V` in `L`, operating recursively if `V` is an array.

```
// A
context 'Assign(HOLE,_)
// B
context 'Assign(_:KResult,HOLE)
// C
rule
  'Assign(R:Ref => convertToLoc(R),_) [intermediate]
// D
rule
  'Assign(L:Loc,V:Value) => copyValueToLoc(V,L) ~> V [step]
// E
rule
  'Assign(_:Result,V:ConvertibleToLoc => convertToLoc(V,r))
  when notBool isLiteral(V)
  [intermediate]
// F
rule
  'Assign(L:Loc, L1:Loc) => reset(L) ~> 'Assign(L,L1)
  when currentOverflow(L1)
  [intermediate]
// G
rule
  'Assign(L,L1) => 'Assign(L,converttoLanguageValue(L1))
  when notBool currentOverflow(L1)
  [intermediate]
```

Figure 5.4: Semantic rules for assignment.

If the RHS is not a value, then rule (E) forces it to be converted to a location. If the location `L1` thus obtained contains an array whose *current pointer* is overflown (e.g. after repeatedly using `next` as in the example below), the current pointer of the assignment target `L` is reset by rule (F). In the remaining cases, rule (G) converts the location `L1` to a value, enabling rule (D) to finally perform its task.

A digression: assignment, arrays and copy-on-write

During the development of the semantics, and motivated by testing and experimenting with the Zend engine, we concluded that, when assigning `$y = $x` where `$x` is an array whose current pointer is overflown, the current pointer of `$y` will be reset to its initial position, as described below. Consider the program

```
$x = array(0);
// let the current overflow
```

```
next($x);
// the aliasing here..
$z = &$x;
// ... forces this assignment to be executed istantly (no copy-on-write)
$y = $x;
// $y was reset, but not $x, which makes sense,
echo "y:␣" . current($y) . "\n";
echo "x:␣" . current($x) . "\n";
```

whose execution will print `y: 0 x: `, meaning that `$y`, the destination variable, was reset, but not `$x`. This is the expected behaviour.

However, it is not always the case that the destination variable (i.e. the left-hand-side) is reset. Consider instead a variation of the above program

```
$x = array(0);
// let the current overflow
next($x);
// $x is not really copied, but a reference assignment is made
$y = $x;
// now a copy(-on-write) is made.
// However, $x is reset (since it is the one named in the instruction,
// and the engine lost track of who was LHS/RHS)
$x[0] = 123;
// this time, $x was reset!
echo "y:␣" . current($y) . "\n";
echo "x:␣" . current($x) . "\n";
```

This time the program prints `y: x: 0` , meaning that `$x` was reset but not `$y` - the opposite of the above example and definitely not the intended behaviour.

This has been filed as a bug report [113] (by other PHP users), yet the issue has never been fixed nor properly understood. We believe that the intended semantics for assignment is the one we adopted, including the fact that when the right-hand-side of the assignment is an array whose current pointer is overflown, the current pointer of the destination is reset to its initial position.

Our hypothesis is that the deviant behaviour is caused by the presence of copy-on-write. Since there is no aliasing on `$x` when the assignment `$y = $x` is performed, the copy is delayed due to copy-on-write. The instruction immediately following the assignment is `$x[0] = 123;`, which updates `$x` and therefore, according to copy-on-write, causes the delayed copy (or "split") to take place. According to the intended semantics, since the current pointer of `$x` was overflown, the corresponding pointer of `$y` would also need to be reset as part of the delayed copy operation. However, at this point, the engine has lost track of the relationship between `$x` and `$y` in the (already executed) assignment (i.e. which was on the left/right hand side) and therefore the reset operation is performed on `$x` - the variable involved in the update. If `$y` was updated instead of `$x`, we would get back the intended behaviour. We believe ours is a satisfactory explanation of the issue, however we do not model this behaviour which we consider to be a bug in the Zend engine.

Array copy semantics

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

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

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

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

and the outputs:

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

and

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

These results show that array `$x` is copied element by element in `$y`, so that the assignment to `$x[0]` affects only `$x`, *except* for the aliased element `$x[1]`, which is now shared with `$y`, which therefore also sees the side effects of the second assignment. Accordingly, our semantics copies the shared elements of the array by reference, and the non-shared elements by value. If a non-shared element is an array itself, the process continues recursively.

This behaviour is defined by the `copyValueToLoc` internal operation, already mentioned in 5.2.4 while discussing the semantics of assignment. `copyValueToLoc` takes two arguments, a value and a location, and copies the value into that location. In case the given value is an array, the copy is performed as discussed above, by distinguishing between the case in which an array element is shared or not. Moreover, since array elements could be arrays themselves, the mechanism has to be repeated recursively every time this happen. The definition of `copyValueToLoc` is therefore split into two main cases. When the input value is a scalar, the definition is simple

```
rule [copy-value-to-loc-scalar]:
  <k> copyValueToLoc(V:LanguageValue, L:Loc) =>
  write(V, L) ... </k>
  when notBool (isArrayType(V) orBool isObjectType(V))
  [internal]
```

and essentially relies on the `write` low-level operation to simply write the value in the desired location.

Conversely, when the input value in an array, `copyValueToLoc` first stores an empty array into the destination location, and then calls `copyArrayMembers` in order to populate it with copies of the original array's elements:

```
rule [copy-value-to-loc-array]:
  <k> copyValueToLoc(Array(Crnt,Elems), L:Loc) =>
  ArrayCreateEmpty(L) ~>
  copyArrayMembers(L,Elems) ~>
  ArrayInitCrnt(L, ArrayIndexOf(Crnt,Elems))
  ... </k>
  [internal]
```

`copyArrayMembers` operates by considering each element in the origin and copying it into the destination, performing a recursive call when the considered element is an array itself. Moreover, for each element considered, the copy is either performed by value or by reference, depending on wether that element is aliased or not. This fact is simply detected by inspecting the element's `isRef` flag, which is `true` in the presence of aliasing and `false` otherwise. Let us first consider the case in which the considered key is not aliased. In this case, the copy must be performed by value

```
rule [copyArrayMembers-cons-no-aliasing]:
  <k> copyArrayMembers(L1:Loc, ListItem([Key,Visib,L]) Elems) =>
    ArrayCreateElem(L1,Key,Visib,allocValue(V)) ~>
    copyArrayMembers(L1, Elems) ... </k>
  <heap> ...  L |-> zval(V:Value,_,_,false) ... </heap>
  [intermediate]
```

and this is accomplished in our rule by relying on `allocValue` to copy the value into a fresh location (operating recursively in case it is a compound value). In the presence of alising instead, the destination element is associated to the location `L1` of the same element in the origin array:

```
rule [copyArrayMembers-cons-aliasing]:
  <k> copyArrayMembers(L1:Loc, ListItem([Key,Visib,L]) Elems) =>
```

```
    ArrayCreateElem(L1,Key,Visib,L) ~>
    copyArrayMembers(L1, Elems) ... </k>
  <heap> ...  L |-> zval(V:Value,_,_,true) ... </heap>
  [intermediate]
```

Notice that the affected reference counters are automatically incremented by `ArrayCreateElem`, which takes care of inserting the given kay into the array at the given location. Finally, notice that, in `copyValueToLoc`, a final operation `ArrayInitCrnt` is invoked after the array has been copied. This simply adjusts the destination array's `current` pointer to match the one of the origin (since when building the destination, its `current` is set to a default value and therefore need to be adjusted manually after the copy).

It is probably worth mentioning at this stage that the behaviour we have just discussed is not included in Facebook's PHP specification, which instead regards it as an "implementation-dependent" choice [75]. To quote the specification itself: "If `$source`'s `VStore`[2] has a refcount that is greater than `1`, the Engine uses an implementation-defined algorithm to decide whether to copy the element using value assignment (`$destination = $source`) or byRef assignment (`$destination =& $source`)."

Evaluation order

In C, the evaluation order of expressions is undefined, i.e. different behaviours can be observed. In most languages, it follows a left-to-right order. Let us consider what happens in PHP.

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

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

[2]In Facebook's PHP specification, `VStore` is essentially equivalent to our `Zval`.

```
echo $c; // prints "twotwo"
```

This example suggests that the operands of the string concatenation operator "." are indeed evaluated left-to-right. Unfortunately things are not that easy. Let us see what happens if the operands are simple variables instead of array elements:

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

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

Both print `"twotwo"`, contradicting our hypothesis: the example on the left suggests that expressions are evaluated right-to-left. The issue has been reported and discussed on the online PHP bug tracking system [115] but never explained, fixed or resolved by the PHP language developers.

Through the lenses of our formal semantics we are able to fully explain this behaviour as follows. The arguments to binary operators are indeed evaluated left-to-right, but while evaluating array elements (or object properties) yields the corresponding value, evaluating simple variables yields a pointer that is dereferenced only when the value is effectively needed. In this case, the value of `$a` to the left of the string concatenation is read only after the assignment to the right has taken place, which explains the apparently odd behaviour observed in the examples above.

We fully model this behaviour in our rules for binary arithmetic and string concatenation operations, which allow us to pass all of the Zend tests which explicitly target those use-cases. To give the reader an idea about how our rules are structured, consider the string concatenation operation. First, a set of `context` rules enforces the desired (left-to-right) evaluation order:

```
context 'Concat(HOLE,, _:K)
context 'Concat(_:BasicRef,, HOLE)
context 'Concat(_:LanguageValue,, HOLE)
```

In particular, the first argument (i.e. the one on the left) is always evaluated to a reference, in case it was not a literal value (first `context` rule). Then, the second argument (i.e. the one on the right) is also evaluated to a reference, but *only* if the first argument has been reduced to a language value or a `BasicRef` (which essentially denotes simple variables - arrays and objects are evaluated to `lref`s instead). In summary, the `context` rules above evaluate the first argument and then evaluate the second but only if the first is not an array element. When this happens, i.e. when the first argument is an array element, the following rule causes its complete evaluation to a language value *before* the second argument is even considered

```
rule [concat-LHS2LangValue-1]:
  <k> 'Concat((R:LRef => convertToLanguageValue(R)),,_) ... </k>
  <trace> Trace:List => Trace ListItem("concat-LHS2LangValue-1") </trace>
  [intermediate]
```

Since the first argument (which is an array element in this case) has been fully evaluated (to a language value), the possible side effects of the second argument will not affect it. Afterwards, similar rules will cause the second argument to be evaluated. If instead the first argument was *not* an array element, the `context` rules above would have caused the second argument to be evaluated to a reference *before* the first argument is fully evaluated, so that any side-effect of the second argument may also affect the final value yield by the first argument. Finally, once both arguments has been reduced to references, those are resolved to language values and the string concatenation is performed.

Also this behaviour is left out of Facebook's PHP specification, and instead considered implementation-dependent [75].

Functions

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

by value or *by reference* and its value is fixed at declaration time by appending `&` to a function's name. A function call is parsed in the AST as `'FunctionCall(E,Args)`. Expression `E` is evaluated to a string `FN` (the function name), used to retrieve from the `functions` cell the various parameters described above. The reason why the function name is given as an expression (as opposed to a literal string) is to allow the behaviour known as "variable functions":

```
function foo() {
        echo "hello";
}
$function_name = "foo";
$function_name();       // prints "hello"
```

Once the function is retrieved, execution continues by placing at the front of the `k` cell the internal `runFunction` term shown in the rule below, which replaces the contents of the cell `k` with a new list of internal commands (the current program continuation `K` will be saved on the stack).

```
rule [runfunction]:
  <k> runFunction(FN:String,f(FP:K,FB:K,RT:RetType,LS:Loc),Args:K) ~> K  =>
    processFunArgs(FP,Args) ~>
    pushStackFrame(FN,K,L,CurrentClass,CurrentObj,RT,D) ~>
    ArrayCreateEmpty(L1) ~>
    setCrntScope(L1) ~>
    incRefCount(L1) ~>
    copyFunArgs ~>
    FB ~> 'Return(NULL) </k>
  <currentScope> L:Loc </currentScope>
  <class> CurrentClass:Id </class>
  <object> CurrentObj </object>
  <functionArgumentDeclaration> D => . </functionArgumentDeclaration>
  when fresh(L1)
  [internal]
```

The first command evaluates the function arguments `Args` in the current scope. This depends on the

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

Method call

Calling a method is similar to calling a function, with the difference that while a function is always available and can be called from any context, methods are attached to a particular class or object, and can only be called under particular circumstances - i.e. when they are *visible* according to a set of *visibility rules*. In our semantics we perform those checks and then, if the method *can* be accessed, we rely on the existing function call mechanism. If the method cannot be accessed we report a runtime error.

The process is initiated by the following rule (we omit auxiliary rules whose role is to pre-process the arguments)

```
rule [meth-call]:
  <k> 'FunctionCall(Obj:Loc,,MethodName:String,,'ListWrap(Args:KList)) =>
       methRunIfVisible(
         methodLookup(ObjClass,MethodName),'ListWrap(Args),Obj) ... </k>
  <heap> ... Obj  |-> zval(OID(_,ObjClass:Id,_),_,_,_) ...  </heap>
```

Notice how, in order to find the required method, Obj is matched in memory so that its class name is retrieved. This information, alongside Obj itself, the arguments and the method name is then passed as argument to the internal operation methRunIfVisible, which is responsible for calling the method or raising an exception, depending on wether the method is accessible from the current context. In turn, methRunIfVisible relies on methodLookup, an internal function that, given a class name and a method name, finds the method in the class hierarchy and returns a

descriptor or an error code if the method is not found. If the method is found, and depending on the outcome of the visibility checks, `methRunIfVisible` will either call the method or report an error. `methodLookup` is defined recursively. In the base case, the method name is found in the class given as argument. In this case no further search is necessary and a descriptor containing the information necessary to call the method is returned:

```
rule [meth-lookup-found]:
  <k> methodLookup(ClassName:Id,MethName:String) =>
      methodInfo(ClassName,MethName,Visibility, Static) ... </k>
  <classes> ... ClassName |-> class(_,_,Meths:Map(MethName |->
      method(_,Visibility:ArrayItemVisibility,Static:Bool)),_) ... </classes>
```

Otherwise, if the method is not found in the specified class, the search continues in the parent class:

```
rule [meth-lookup-parent]:
  <k> methodLookup(ClassName:Id, MethName:String) =>
      methodLookup(Parent, MethName)  ... </k>
  <classes> ... ClassName |-> class(Parent:Id,_,Meths:Map,_) ... </classes>
  when notBool(MethName in keys(Meths))
  [internal]
```

Finally, if we reach the root of the class hierarchy without finding the method, the program is aborted and an error is reported (we discuss the mechanism of errors and warnings in more detail later in this section):

```
rule [meth-lookup-not-found]:
  <k> methodLookup (ClassName:Id,MethName:String) =>
      ERROR(
        makeAccess2NonDefinedClassMethodErrorMsg(ClassName,MethName))  ... </k>
  when Id2String(ClassName) ==String "ROOT"

// construct error message
rule [make-undef-class-meth-error]:
  makeAccess2NonDefinedClassMethodErrorMsg(Class,Var) =>
```

```
  "Call to undefined method "  +String
    Id2String(Class) +String "::" +String "$"
    +String Var +String " in " +String "%s" +String
    " on line " +String "%d\n"
```

In case `methodLookup` found a method and returned a descriptor, `methRunIfVisible` checks the visibility and run the method if the constraints are met:

```
rule [meth-run-if-visib-non-static]:
  <k> methRunIfVisible(methodInfo(Class:Id,
      Name:String, Visibility:ArrayItemVisibility, false), Args:K, Obj) =>
      #if isVisible(Class, CurrentClass, Visibility) #then
        methExecute(Class, Name, Args, Obj)
      #else
        ERROR) ... </k>
  <class> CurrentClass:K </class>
```

The last rule relies on the answer provided by the `isVisible` predicate. If the true branch is taken, control passes to the `methExecute` internal operation (omitted for brevity) which passes the relevant information to the existing function call (internal) rules (such as `runFunction` shown while discussing function calls). Otherwise an error is reported. The `isVisible` predicate is defined in a standard way and omitted for brevity.

Static Methods

Calling a static method is very similar to calling a non-static one. Indeed, the underlying machinery, i.e. the function call rules, are the same. The main difference is that in this case we don't have an object in our context, since we are calling the method statically. This can be seen in how the AST node is presented:

```
'StaticFunctionCall(ClassName:String,,MethName:String,,'ListWrap(Args:KList))
```

there is no `Obj` argument as in the non-static method call. Instead, there is a class name, the name of the class whose static method we are willing to call. To handle this we pass the information to

the methRunIfVisible internal operation, already discussed in the context of method calls:

```
rule [static-fun]:
  <k> 'StaticFunctionCall(ClassName,,MethName,,'ListWrap(Args:KList)) =>
      methRunIfVisible(methodLookup(String2Id(ClassName), MethName),
      'ListWrap(Args), none) ... </k>
  when isKResult(Args)
  [step]
```

The only difference with standard method call is that the last parameter to methRunIfVisible, i.e. the object whom method is being called, is none. methRunIfVisible will check if indeed the method has been defined as static and then pass the information to methExecute which will finally call runFunction to call the method. However, runFunction will receive no object parameter, and for this reason it will not update the <currentObject> cell (as done in standard function call). As a result, calling a method statically does not introduce a new binding for $this. If the method is called from top-level, there will be no $this, otherwise, its existing binding, if present, will be maintained. However, note that the currentClass part of the context information is still updated to reflect the name of the class defining the called method.

There are further cases for dealing with non-static methods called as static, static methods called as non-static etc., where we basically throw a warning and try to desugar the call to a standard case. Those cases are available in the source code, but we avoid presenting them here.

Loops and Break

We define each looping construct (e.g. while, for and foreach) independently using the same techniques discussed so far. However, in order to allow break and continue, we also maintain a *loop stack* and implement a general strategy for dealing with loops which we outline here.

Every time a loop is encountered, an item (which we call a "loop frame" in $\mathbb{KPHP}$) containing the code immediately following the loop is pushed into the loop stack. Then the loop is executed (according to its semantics), followed by a special popLoop operation. If no break or continue are encountered (and the loop terminates), the popLoop operation is eventually reached and evaluated,

causing the previously pushed frame to be destroyed. In this case, execution continues as normal. If `break` or `continue` is encountered in the loop body, the previously pushed frame is used to retrieve the fragment of program which follows the loop. The current computation is discarded and replaced by the one found in the frame so that control can jump to the program point immediately following the loop.

For the reasons above, all looping rules share the same high-level structure. Let `'L` be the AST node for a looping construct. Then, we give rules in the form

```
'L (args) ~> K => PushFrame(K) ~> L(args) ~> popLoop
```

Where `L` (note there are no single quotes) is the (internal rule) that effectively define the construct represented by AST node 'L.

We now show the semantics rules for `break`. The construct expects an integer argument, the numbers of nested enclosing structures to be broken out of. We define the behaviour recursively in the (integer) argument. The base case (i.e. when the argument is 1)

```
rule [break-1-normal]:
  <k> 'Break(1) ~> _ => K </k>
  <loopStack> ListItem(loopFrame(K,none)) => .List ... </loopStack>
  [step]
```

behaves exactly as discussed above. It discard the current computation, replacing it with the one found in the `loopFrame`, before discarding the frame itself. The recursive case

```
rule [break-n]:
  <k> 'Break(NSteps => NSteps -Int 1) ... </k>
  <loopStack> ListItem(K) => .List ... </loopStack>
  when NSteps >Int 1
  [intermediate]
```

simply consumes the current frame and decrement the argument by one, until the base case is reached. We should also mention that, in addition to the simple cases above, we also had to define a corner-case to deal with `foreach`. We detail this when we discuss `foreach` and its challenges later in this section.

To show the mechanism in action, consider for example the `while` loop. In our semantics, it is modelled in terms of the conditional statement via a well-known equivalence result [10] stating that the two programs

```
while G {B};
```

and

```
if G
        B; while G {B};
```

are *equivalent*. Ignoring the discussion above about `break` and the needed machinery, our semantics rule would be the following:

```
rule [while]:
  <k> 'While(Condition:K, Body:K) =>
      'If(Condition,, Body; 'While(Condition, Body )) ... </k>
  [structural]
```

In order to support `break`, we instead change the above rule as follow:

```
rule [while]:
  <k> 'While(Cond:K,, Body:K) ~> K:K =>
       pushLoopContext(loopFrame(K,none)) ~>
       while(Cond, Body) ~>
       popLoopContext ... </k>
  [step]
```

where the content of the original rule, giving actual semantics to `while`, has been moved into an auxiliary rule `while-spec`:

```
rule [while-spec]:
  <k> while-spec(Condition:K, Body:K) =>
      'If(
        Condition,,
        Body; while-spec(Condition, Body),,
      ) ... </k>
  [structural]
```

Other simple control-flow constructs such as `if` and `for` have a standard semantics (usually borrowed from standard tutorial K definitions, modulo the remarks above about the fact that we maintain a special stack for loops), therefore we avoid discussing them here (as usual, the interested reader is referred to `http://phpsemantics.org`). Instead, we discuss in great detail the `foreach` construct, which turned out to be challenging to model. Other authors has chosen not to model similar constructs at all; for example, in JSCert (see Section 2.3) the construct is not modelled due to its non-determinism.

Object and array iteration (foreach)

Scripting languages such as JavaScript and PHP provide constructs to iterate over all the properties of an object. Such constructs can be tricky to implement, and hard to model in a formal semantics. For example, the JavaScript formalisation of [20] does not model the for-in loop, as the corresponding ECMA5 standard is inconsistent when it comes to describe object updates within the loop. We discovered that the corresponding `foreach` statement in PHP is also very challenging, due to the presence of aliasing and the behaviour of the explicit `current` pointer.

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

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

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

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

This is what happens: variable `$v`, introduced by the first `foreach`, has global visibility; at the end of the first `foreach`, `$v` and `$a[2]` are aliased; at every iteration of the second `foreach`, a simple assignment `$v = $a[...]` is made, storing the current array element in `$v`, and hence in `$a[2]`.

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

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

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

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

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

This code enters an infinite loop outputting `1,3,3,3,3...`. Since PHP arrays are copied by value (as opposed to objects which are copied by reference), the assignment `$a1 = $a2` copies the whole data structure associated to `$a2`, including its `current` pointer, which is used to perform the iteration, causing the infinite loop (see 5.2.4 for more details on array assignment).

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

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

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

As just shown, the `foreach` construct is tricky to implement and analyse due to the corner cases that arise in the presence of aliasing. While, in principle, it could be defined in terms of simple looping (e.g. `while`) and the `current` and `next` array operators, the above corner cases made it impossible for us to model it that way. Instead, we adopted a *hybrid* approach where we essentially translated `foreach` into a `while` loop whose body is made of a mix of language constructs and internal (semantics level) operations.

To start with, `foreach` expects its first argument, that is the array or object to be iterated, to be a location:

```
rule [foreach-arg2Loc]:
  <k> 'ForEach((R:ConvertibleToLoc => convertToLoc(R,r)),,_:K,,_:K) ... </k>
  [intermediate]
```

We now discuss the rules that are applied next after the argument has been successfully evaluated to a location. For all other cases (e.g. the argument was a scalar) an error or warning is reported.

To be able to correctly model behaviours such as those discussed in 5.2.4, two cases need to be considered once the iterator argument is reduced to a location. If the argument is not aliased, the actual iteration is performed on a *local copy* of the array or object and the following steps are taken:

- copy the argument `L` into `Lx`,
- push a loop frame into the loop stack. Notice it contains a term `foreachArrayPair(L1:Loc,L2:Loc)` encapsulating two locations, the second of which is optional. We clarify this term later.
- call the internal operation which actually performs the `foreach` iteration
- pop the loop frame from the stack

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

The following two rules perform the sequence of operations outlined above with a slight variation depending on wether the iterator is an array or an object. The two rules differ in the way they construct the loop frame to be pushed in the stack, and could in principle be factorised by inserting a (semantics-level) conditional in the relevant part of the rule.

```
rule [foreach-with-local-copy-array]:
  <k> ('ForEach(L:Loc,,Pattern:K,,Stmt:K) ~> K:K) =>
        write(V,Lx) ~>
        pushLoopContext(loopFrame(K, foreachArrayPair(L,Lx))) ~>
        foreach(Lx, Pattern, Stmt) ~>
        popLoopContext
  </k>
  <heap> ... L |-> zval(V:Array,_,N,_) ... </heap>
  <currentForeachItem> _ => L </currentForeachItem>
  when ((V isCompoundValue) andBool (N <=Int 1) andBool (fresh(Lx:Loc)))
  [step]

rule [foreach-with-local-copy-object]:
  <k> ('ForEach(L:Loc,,Pattern:K,,Stmt:K) ~> K:K) =>
        write(V,Lx) ~>
        pushLoopContext(loopFrame(K, foreachArrayPair(L, none))) ~>
        foreach(Lx, Pattern, Stmt) ~>
        popLoopContext
  </k>
  <heap> ... L |-> zval(V:Object,_,N,_) ... </heap>
  when ((V isCompoundValue) andBool (N <=Int 1) andBool (fresh(Lx:Loc)))
  [step]
```

The `foreachArrayPair` term is used by `popLoopContext` to recognise the case where a `foreach` operation has been performed on a local copy of the iterated value (array or object) and therefore, before terminating the loop, it is necessary to update the original copy's `current` pointer in order to give the user the impression that the iteration has been performed on the original argument. The term `foreachArrayPair(L,Lx)` left on the stack by the first rule ensures the `current` pointer

of the array in `Lx` is copied into the array in `L` before exiting the loop. However, in the second case, when the iterator is an object, this operation is not necessary (because objects are copied by reference anyway), so the term `foreachArrayPair(L,none)` is left into the stack, indicating there is no need to update the `current` pointer of the object in `L`. This difference is due to how arrays and objects are copied in PHP. While arrays are copied by value, which makes the update necessary, objects are copied by reference (by copying the OID containing the pointer to the array containing the fields) and therefore manipulation of the `current` pointer of the copy influence the `current` pointer of the original object as well (see 5.2.1).

In all other cases, i.e. when the iterator is either an array or an object *and* it is aliased at the moment the `foreach` is called, no local copy is required and the internal `foreach` rule is called directly:

```
rule [foreach]:
  <k> ('ForEach(L:Loc,,Pattern:K,,Stmt:K) ~> K:K) =>
        pushLoopContext(loopFrame(K, foreachArrayPair(L, none))) ~>
        foreach(L, Pattern, Stmt) ~>
        popLoopContext
  </k>
  <heap> ... L |-> zval(V:Value,_,N,_) ... </heap>
  when ((V isCompoundValue) andBool (N >Int 1))
  [step]
```

Finally, the `foreach` internal operation performs the actual array or object iteration. For readability, we first present a pseudo-code version of the procedure, followed by the actual $\mathbb{K}$ rule. The semantics of `foreach` (in the variants `foreach ($x as $v)`, `foreach ($x as $k => $v)` and `foreach ($x as $k => &$v)`) is described at high-level as follows:

1. `reset` the array or object being iterated
2. fetch the current element, and store it in `crnt`
3. while (`crnt` is not `NULL`)

(a) if the current element is visible according to the visibility rules

i. asssign the current element (and optionally its key) to the iteration variables, as specified in the pattern. This can be either

A. `foreach ($x as $v)`

B. `foreach ($x as $k => $v)`

C. `foreach ($x as $k => &$v)`

ii. advance the current pointer by one position

iii. fetch the next current element and store it in `crnt`

iv. execute the loop body

v. if the iterator does not contain a compound value anymore (due to side-effects of the loop body) then execute a `break`.

(b) otherwise (i.e. in case the current element is not visible from the current context)

i. advance the current pointer by one position

ii. fetch the next current element and store it in `crnt`

The following $\mathbb{K}$ rule implements the algorithm above.

```
rule [foreach-spec]:
  <k> foreach(L:Loc, Pattern:K, Stmt:K) =>
    'ListWrap(
    'Expr(reset(L)),,
    'Expr('Assign('Variable('Simple("crnt")),,key(L))),,
    while('IsNotIdentical('False(.KList),,'Variable('Simple("crnt")))),
    *Cond(
      isCrntVisib(L),
      'ListWrap(
        initIterationVars(L,Pattern) ~>
        'Expr(myNext(L)) ~>
        'ListWrap('Expr('Assign('Variable('Simple("crnt")),,key(L)))) ~>
        Stmt,,
```

```
        'If(
          locHoldsCompoundValue(L),,
          'ListWrap(.KList),,
          'ListWrap(.KList),,
          (WARNING("Invalid argument supplied for foreach() in %s on line %d\n") ~> 'Break(1)
        'Expr(myNext(L)) ~>
        'ListWrap('Expr('Assign('Variable('Simple("crnt")),,key(L))))))) ... </k>
  <heap> ... L |-> zval(V:Value,_,_,_) ... </heap>
  <currentForeachItem> L1:Loc </currentForeachItem>
  <trace> Trace:List => Trace ListItem("foreach-spec") </trace>
  when (V isCompoundValue)
  [internal]
```

We omit the description of the auxiliary operation `locHoldsCompoundValue(L)`, which performs a simple check to decide wether the location `L` contains to a compound value or not. The auxiliary operation `initIterationVars(L,Pattern)` takes care of assigning the current element of the array or object to the iteration variable(s) as specified by the pattern for each of the three variants allowed by the language.

In contrast, Facebook's PHP specification only provides a very general description of the intended behaviour of `foreach` and fails to capture the subtleties that we have discussed and successfully modelled in our semantics. According to the specification [75]: *"The foreach statement iterates over the set of elements in the collection designated by foreach-collection-name, starting at the beginning, executing statement each iteration. On each iteration, if the* `&` *is present in foreach-value, the variable designated by the corresponding expression is made an alias to the current element. If the* `&` *is omitted, the value of the current element is assigned to the corresponding variable. The loop body, statement, is executed zero or more times. After the loop terminates, the variable designated by expression in foreach-value has the same value as it had after the final iteration, if any.*"

Exceptions

Similarly to loops and `break`, the treatment of exceptions relies on an *exception stack* which contains a list of *exception frames*. When encountered, the `try-catch` statement is processed as follows. First, an exception frame is built using information from the statement as well as from the configuration, and pushed onto the exception stack; the original `try-catch` statement is then replaced with the `try` body followed by the special `popx` internal operation:

```
rule [try-catch]:
  <k>
    ('Try('Body(TryBody:K),,
        'ListWrap('Catch(XClass:String,,VarName:K,,'Body(CatchBody:K)))) =>
    TryBody ~> popx) ~> K:K
  </k>
  <control>
    C:Bag
    <exceptionStack>
      . => ListItem((String2Id(XClass), VarName, CatchBody, K, C)) ...
    </exceptionStack>
  </control>
  [step]
```

At this point control is passed to the `try` body which is executed until either the `popx` operation is reached or an exception is thrown.

In the first case no exceptions have been thrown, and the `popx` operation takes care of discarding the exception stack frame previously created. The execution then continues from the next instruction.

```
rule [pop-exception-frame]:
  <k> popx => . ... </k>
  <exceptionStack> _ => . ... </exceptionStack>
  [internal]
```

The other possibility is that a `throw` statement is encountered *before* the `popx` operation. In this

case, the `catch` body together with the variable that needs to be bound to the exception itself and the previous state information are retrieved from the exception frame, which is then discarded. The context information is restored, the exception value assigned to the designated variable and the control finally passed to the `catch` body followed by the remainder of the original computation:

```
rule [throw]:
  <k> 'Throw(V:Object) ~> _ => 'Expr('Assign(VarName,,V)) ~> CatchBody ~> K </k>
  <control>
  <exceptionStack>
    ListItem((XClass:Id, VarName:K, CatchBody:K, K:K, C:Bag)) => . ...
  </exceptionStack>
  (_ => C)
  </control>
  [step]
```

A similar rule (which is omitted for brevity) deals with the case where a `throw` statement is encountered but the exception stack is empty. In this case, an *uncaught exception* fatal error is raised and the program is aborted.

Static class variables

Access to static class variables (as well as to static methods), is done via the *scope resolution* operator. The following example shows how a static class variable can be declared and accessed - note that the class does not need to be instantiated.

```
class MyClass {
        // ...
        public static $a = 'constant value';
        // ...
}
MyClass::$a;
```

In our semantics this operation is modelled by the `'ClassConstant(ClassName,VariableName)` operation. Since our representation of classes explicitly keeps static variables apart from instance

variables, the task amounts to finding the data structure associated to `ClassName`, accessing its *static variables storage space* and finally looking for `VariableName` there, taking into account the usual visibility constraints and the possibility that the variable might be defined (hence stored) in a superclass. This last case is dealt with by the following rule, which rewrites its first argument, `ClassName`, with the name of `ClassName`'s parent.

```
rule [class-constant-parent]:
  <k> 'ClassConstant((ClassName:Id => Parent),,'Variable('Simple(X:String))) ... </k>
  <classes> ... ClassName |-> class(Parent,_,_,L)  ...  </classes>
  <heap> ... L |-> zval(Array(_,M),_,_,_)  ... </heap>
  when notBool (M hasProperty X)
  [intermediate]
```

Eventually, either the variable name is found, or `ROOT` is reached. In the latter case we simply issue an error while in the former we return the location in which the variable is stored, but *only if the variable is visible* from the current context, according to the visibility rules (as discussed before). The following $\mathbb{K}$ rule implements this strategy.

```
rule [class-constant]:
  <k> 'ClassConstant(ClassName:Id,,'Variable('Simple(X:String))) =>
       *Cond(
         isVisible(ClassName,Context,V),
         L1,
         ERROR(makeAccess2NonVisibleClassConstantErrorMsg(ClassName, X, V))) ...
  </k>
  <classes> ... ClassName |-> class(_,_,_,L)  ...  </classes>
  <heap> ... L |-> zval(Array(_,_ ListItem([X,V,L1]) _),_,_,_)  ... </heap>
  <class> Context:K  </class>
  [step]
```

Essentially, the rule retrieves the location in which the desired variable `X` is stored, by matching it inside `ClassName`'s static variables storage (stored in the location `L`). Then, it checks wether the variable is visible according to the current context via the `isVisible` predicate. If `isVisible`

evaluates to `true`, then the location `L1`, containing the static variable, is returned; otherwise an error is reported.

Input and Output

In order to deal with input and output, we rely on the builtin functionality provided by the K framework. In particular, K allows the definition of special *input and output* cells in the language configuration which can be connected to the system's standard input and output via the `stream` attribute. For example, by adding the following line in `configuration.k`

```
<out stream="stdout"> .List </out>
```

we equip our semantics with an output cell `<out>` containing a list and representing an *output stream*. However, the `stream="stdout"` attribute has the effect of attaching a special behaviour to this particular cell: every time a list element is inserted into it, instead of being added to the list as usual, it is immediately printed on `stdout`.

On top of this mechanism, we define all our I/O constructs. In particular, we define a low-level `print` operation as follows.

```
rule [internal-print]:
  <k> print (X)  => . ... </k>
  <out> ... . => ListItem(X) </out>
  [internal, output]
```

When executed, `print` simply causes its argument to be printed on `stdout`. Other higher-level constructs such as `echo` and `var_dump`, which we do not detail here, are defined in terms of this printing primitive.

HTML

In general, a PHP script can be an HTML document that contains several PHP tags `<?` ... `?>` (or `<?PHP` ... `?>`) delimiting regions of PHP code that are executed as part of the same script. The HTML is treated as part of the program output, in the order it is encountered. Our semantics

implements this behaviour. Hence, the `hello.php` example of Section 3.2 is equivalent to

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

Errors and Warnings

PHP, at least in the Zend implementation, has "many levels of errors" which can be enabled or disabled in different combinations via the `error_reporting` internal function (`http://php.net/manual/en/function.error-reporting.php`). As shown in the online documentation (e.g.: `http://php.net/manual/en/errorfunc.constants.php`), there are currently 15 kinds or errors.

Our semantics currently models a subset of them (mostly those that we found necessary in order to successfully pass the Zend test suite) and does not allow enabling or disabling them. We always display all errors and warnings, which help us during testing and debugging. Allowing selective enabling or disabling of one or more categories of errors could be easily achieved by adding a set of flags to the configuration and updating the rules related to error reporting accordingly (i.e. if the flag is set to "no-warning" the rule for warnings simply does nothing, otherwise it perform the expected operations). We leave this for future work. In our development we currently partition the set of errors into *warnings*, *fatal errors*, *notice* and *deprecated*. Additionally we introduced one further category of errors for handling constructs which are not supported by the current version of KPHP.

Our rules for the aforementioned categories of errors all look very similar, the main difference being in the format of the error message they display and in wether the computation is aborted (e.g. for fatal errors) or can continue (e.g. for warning and notices). Furthermore, the rules are instrumented so that they put an integer value into the `<errorManagement>` cell. This value is not part of the PHP semantics, but is used by our test harness (currently written in Python) in order to better organise the produced reports. As an example, consider the rule for warnings:

```
syntax K ::= "WARNING" "(" String ")"
rule [warning]:
 <k> WARNING(Msg:String) => print("\nWarning: ") ~> print(Msg) ... </k>
 <errorManagement> _ => 1 </errorManagement>
```

```
  [internal]
```

Its only effect is to print a message - the computation can then continue. However, the value 1, associated to warnings, has been written into the errorManagement cell, so that in the HTLM report from our test harness the test that triggered the warning will be placed into a "tests with warnings" category.

The rule for fatal errors is similar, with the difference that all remaining computation is consumed - i.e. it replaces itself *and* the following computation with the error message. Notice that this time the error code is 2.

```
syntax K ::= "ERROR" "(" String ")"
rule [error]:
  <k> ERROR(ErrorMsg:String) ~> K:K => print("\nFatal error: ") ~> print(ErrorMsg) </k>
  <errorManagement> _ => 2 </errorManagement>
  [internal]
```

The remaining error management rules are similar.

5.3 Putting it all together: KPHP

As shown in section 2.2.2, we compile and execute our semantics with the kompile and krun tools respectively. However, since we currently rely on an external parser, a path to it must also be provided to krun. We can easily turn our semantics into a PHP interpreter by running the following command:

```
krun --parser="java -jar parser/parser.jar" --no-config test.php
```

where test.php is the file containing the PHP script we wish to evaluate and the option --no-config tells krun to avoid displaying the final configuration. In order to make the process easier and more similar to what one would expect from a standard language interpreter, we wrap the above functionality into a shell script called kphp. The following command is equivalent to the previous:

```
kphp test.php
```

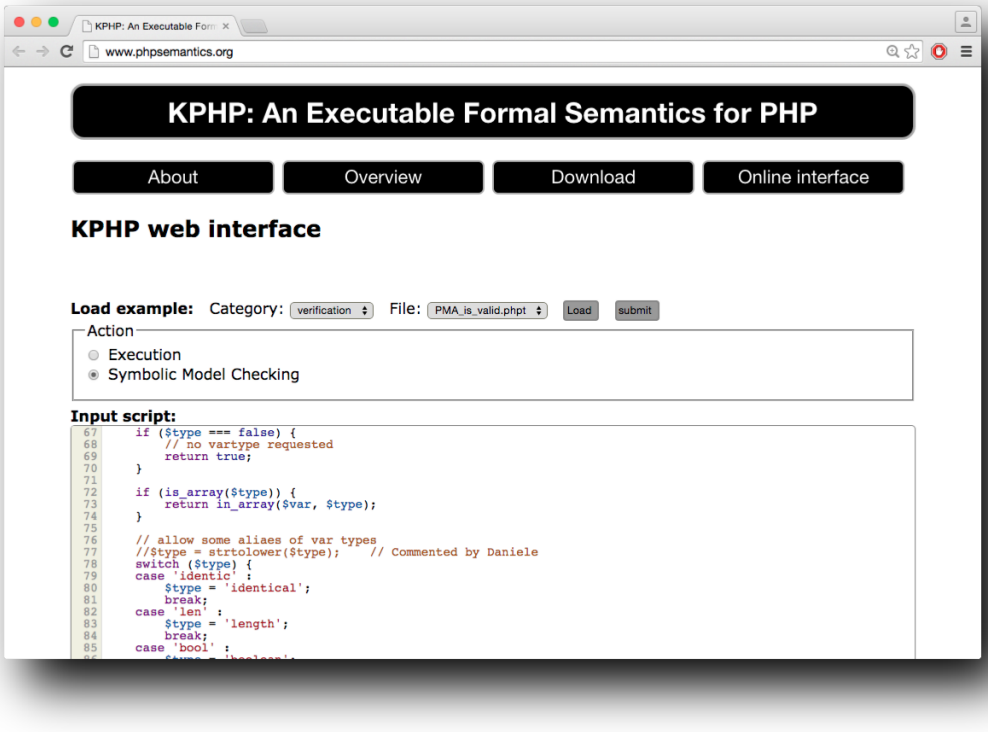


Figure 5.5: A screenshot of the KPHPweb interface available at `http://phpsemantics.org`

If, for debugging, the user wants to display the final configuration she could simply add the `--config` option:

```
kphp test.php --config
```

We also developed a web-interface where PHP programs can be run and model-checked (see chapter 6) using KPHP. The user can either load predefined examples (some from the Zend test suite and some from [17]) or write her own code in the built-in text editor. or A screenshot is shown in figure 5.5. The web interface, together with the full source code and a tutorial is available at `http://phpsemantics.org`.

5.4 Testing and Validation

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

sion of this thesis.

5.4.1 Test-driven semantics development

As discussed in Section 3.2, at the time we started our formalisation there was no official document providing a specification of PHP. On the other hand, the recent PHP specification introduced by Facebook [75] still provides a very abstract memory model and doesn't specify the more involved language features and corner cases, as discussed in 3.2. Hence, the development of our semantics was largely test-driven. The choice of specifying the semantics in $\mathbb{K}$ meant that at each stage of this work we had a working interpreter corresponding to the fragment of PHP we specified up to that point. This made it possible to test critical semantics rules as they were being developed. For this ongoing testing we wrote snippets of PHP, and compared the results from our interpreter with the ones from the Zend Engine, which is the *de facto* reference interpreter. Recently, this approach has been extensively used during the development of $\mathbb{K}$-Java [18] (see 2.3).

5.4.2 Comparison with Facebook's PHP specification

Ultimately, we found a detailed comparison of the PHP specification by Facebook (introduced in 3.2.3) with our semantics to be out of scope in this context. First, those two artefacts are hardly comparable, one being an informal English text and the other being a mathematical specification. Second, the goals and scopes of the projects are different. Our semantics aims at precisely modelling the behaviour of Zend, which is the reference implementation of PHP, focusing on its good parts but also on the bad ones. Facebook's specification instead seems to be aimed at providing a subset of characteristic and behaviours that should be implemented by every PHP runtime. In their choice, however, they have left out many of the challenging (but we believe also characteristic) bits of the Zend implementation of PHP, which instead we spent lot of effort to understand and model.

For example, the following is a partial list of challenging features of PHP that we fully model while Facebook's specification doesn't (regarding the details as "implementation dependent"): the

complex array copy mechanism of Zend; the subtle evaluation order strategies for the arguments of binary operators; the `foreach` construct (which is described in a very simplified manner with no mention to the numerous corner cases and different behaviours in the presence or absence of aliasing - see 5.2.4 for a detailed discussion of those features). Similarly to Facebook's specification, we do not to model the copy-on-write mechanism; notice, however, that our choice is motivated by the fact that a copy-on-write mechanism is not intended as a language feature; instead, it is meant as an *implementation optimisation* which should be completely invisible from the user perspective, and therefore irrelevant for semantics modelling.

5.4.3 Validation

As common in other PHP projects (e.g. HHVM [72] and HippyVM [71]), we validated our semantics/interpreter by testing it against the official test suite distributed with the Zend engine. Although our semantics covers most of the core PHP language (including its challenging features, such as arrays, objects, references, aliasing, exceptions, etc.), the test suite includes many tests that refer to constructs or library functions that we do not yet support. The Zend test suite is already split into folders containing different categories of tests. We tested the semantics against all the tests in the folders `lang` (core language) and `functions`: we did not pick tests manually to avoid introducing bias (as instead it has been done in [56] and [57].

The `lang` folder contains 216 tests and we pass 97 of them, the `functions` folder contains 14 and we pass 4. If a test fails, it is for one of four reasons: (i) our semantics models a feature incorrectly; (ii) a language construct is not supported by our semantics; (iii) the external parser has a bug, and returns the wrong AST; (iv) the external parser does not support some features added to PHP after version 5.0. For each failed test, our test harness shows one of these four categories. Successful tests are partitioned in 2 sets: 71 are automatically recognised as success, 26 are considered successful after manual review of the output (for example, our warning and error messages do not contain source code line numbers, because they are not recorded by the external parser). Another source of false positives, due to the internal representation of floating-point

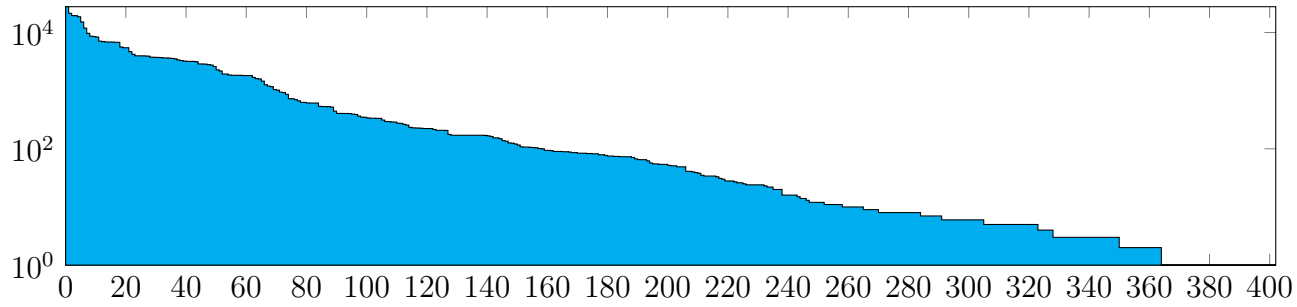


Figure 5.6: Coverage of KPHP rules by the Zend test suite (logarithmic scale).

numbers, was discussed in 5.2.1

The only test that fails in category (i) is `031.phpt`, which tests for the erratic behaviour of `current` described in 5.2.4. This fail is intentional, as we consider such behaviour to be a bug in the Zend Engine. All the other failed tests belong to categories (ii)-(iv), hence are either not supported by the semantics or by the parser. The total number of passed tests may not seem very high (JSCert passes almost 2,000 tests), but this is due to the size of the Zend test suite, and is outside our control. Moreover, many tests are non trivial, focussing on complex evaluation strategies, classes, constructors, interaction between global variables, functions, objects and aliasing. We are satisfied by this validation experiment - so far, our semantics behaves as the official PHP implementation.

5.4.4 Coverage

In order to assess the level of coverage of our semantics achieved by the Zend test suite, we added a `trace` cell to the KPHP configuration, representing a list where we add the name of each rule as it is executed. When running our test harness, we then keep track of the rules being exercised for each program and how many times they are called in total. We then compare this numbers with the set of all defined rules in order to obtain usage statistics. Out of 721 semantics rules, 403 are executed at least once, and 318 are never executed. In Figure 5.6 we show the histogram, ordered by frequency, of the executed rules. There is a big difference in the number of times different rules are exercised by the test suite. This is partly explained by our design. The small group of rules which is called more than $25,000$ times by the test suite corresponds to the low-level, internal rules

which are used as building blocks by other, higher level rules. Internal rules that perform type conversions, such as `*toInt`, are also intensively exercised, as expected. 158 of 403 rules are called at least 100 times, and 264 are called more than 10 times.

Using the Zend tests alone, coverage amounts to 56% of the semantic rules. In order to achieve full-coverage, and in line with out test-driven-development methodology, we have written targeted additional tests (122 in total) that cover the rules not exercised by the Zend suite. As a design choice, we kept our tests short and targeted to a single feature or particular combination of features (as opposed to some of the Zend tests which count more that 200 LOC and attempt to test more than one feature). This approach has been taken a step further in the recent formalisation of Java in $\mathbb{K}$ [18] where the authors produced a set of 840 tests. However, their choice was partly motivated by the fact that they couldn't obtain the Java official test suite (due to licensing issues [18]) and therefore they were strongly motivated to produce their own comprehensive test suite. In our case, since the Zend test suite is available, our goal was to produce additional tests so that the whole semantics, including the rules not exercised by the Zend suite, could be covered[4]. We also plan, in future work, to expand our set of tests and make it public.

5.5 Limitations, Perspective and Future Work

A formal, executable semantics of an actively developed real programming language is too large a task to be completed in one single effort. This thesis models the core of PHP, which includes the features we considered more important and instructive, leaving out some non-core features and the numerous language extensions.

We stress however the practical relevance that this work has had so far. Not only has it been used as a semantics foundation for our verification tools (a symbolic model checker, discussed in Chapter 6, and a general purpose abstract interpreter. discussed in Chapter 7) but it has also inspired numerous Undergraduate and Master student's final year projects and internships.

[4]However, the reader should note that the fact that our test suite covers the whole set of semantics rules doesn't mean, in general, that all possible behaviours are considered.

Examples include an Eclipse-based graphical debugger for PHP [116] and an earlier taint-analysis tool [117]. Similarly, during his 6 months-long internship at Imperial College, a Masters student from Ecole Normale Supérieure (Paris) collaborated with us to the development of a type domain for our general purpose abstract interpreter based on KPHP(discussed in Chapter 7).

In this Section, we summarise what we left out of the current formalisation, and indicate what we think are the next priorities to take this work further.

Parsing limitations. As discussed in Section 5.4, the external parser we currently use does not understand some language constructs introduced after version 5.0, such as for example the literal array syntax with square brackets (`[1,2,3]` instead of `array(1,2,3)`). It also does not parse correctly some constructs such as `$this->a[]` which gets parsed as `$this->(a[])` instead of `($this->a)[]`. In future development, we plan to adopt a fully-compliant external parser.

Missing language features. We have not (yet) implemented a number of non-core language features, and in particular: bit-wise operators, most escape characters, regular expressions, namespaces, interfaces, abstract classes, iterators, magic methods and closures. We do not foresee significant obstacles in integrating these into KPHP. For example, magic methods are special object methods (`__toString, __get, __call`, etc.) with reflective behaviour that are called automatically by PHP. JavaScript has similar reflective methods, and techniques to formalise them are well documented [79, 20]. As a taster, we included in the core language the `__construct` magic method, which is used as a constructor by the `new` command when creating a fresh object and a user-defined constructor is not provided. Since version 5.3, PHP includes anonymous functions, implemented as objects of a special `Closure` class. These are not supported by our parser, but can be easily modelled in our semantics by adding to the object `OID` constructor an optional argument pointing to the entry of the `functions` cell where the anonymous function definition would be stored (using the same mechanism as for regular functions).

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

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

Language extensions. Language extensions, such as the functions that provide access to an SQL database, or that connect a PHP script with a server (and hence with the network) are of fundamental importance for developing web applications, but are squarely beyond the scope of our current work. Our goal is to provide a sound semantic foundation to the core language that glues all such functions together. Until (semi-)automated techniques that help giving semantics to language extensions are developed, our view is that such extensions need to be investigated on a case-by-case basis. Often, they can be abstracted in terms of approximate information such as for example their types, taint behaviour, or side effects, as exemplified by our case study of Section 6.3.2.

Chapter 6

Verification via LTL model checking

In this chapter, we begin demonstrating the practical relevance of our semantics by showing potential applications based on the $\mathbb{K}$-Maude tool chain. In particular, the $\mathbb{K}$ framework exposes Maude's explicit-state LTL model checking to the semantics [19], and supports symbolic execution for any language definition [118]. In the following we show how we used LTL model checking in conjunction with symbolic execution to obtain a PHP code analyser, and we analyse properties of two 3rd-party PHP functions of practical relevance.

6.1 Model checking and symbolic execution in $\mathbb{K}$

A $\mathbb{K}$ definition is eventually translated to a Maude rewrite theory, which can be model checked against LTL formulas using Maude's built-in model checker [119]. In order to use the model checker in a meaningful way with respect to PHP, we need to instrument the semantics in two ways. First, we must decide what semantics rules should be considered as *state transitions* by the model checker, tag such rules, and pass the tags to the `--transition` compilation option. Second, we need to extend LTL with a set of atomic propositions that can be used to express interesting properties of PHP programs.

Symbolic execution has been recently introduced in $\mathbb{K}$ [118] and is enabled by using the option `--backend symbolic` when compiling the $\mathbb{K}$ definition. When symbolic mode is enabled, programs

can be (optionally) given symbolic inputs of any of the types natively supported by the $\mathbb{K}$ tool (`int`,`float`,`string`,`bool`).

6.2 Temporal verification of PHP programs

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

6.2.1 Symbolic values in PHP programs

In order to allow symbolic inputs in PHP programs we introduce an auxiliary construct `user_input`, visible to the programmer as a normal PHP function. Moreover, we add an `<input>` cell to the $\mathbb{KPHP}$ configuration, representing a (possibly empty) *input stream* (i.e. a list of values, symbolic or otherwise). We are then able to pass its initial content to `krun` together with the program to be executed or model checked via the command line. For example, provided the `<input>` cell has been declared as

```
<input> $IN:List </input>
```

in the configuration, we can run a program `test.php` into a state whose input stream contains a symbolic integer and a concrete string `"foo"` with the following command

```
krun -cIN=ListItem(#symInt(x)) ListItem("foo") test.php
```

When invoked inside a program, the auxiliary construct `user_input` simply takes the first item from the `<input>` cell and returns it. The item is then discarded from the `<input>` cell so that the next item will be available for the next call:

```
rule
  <k> user_input => V ...</k>
  <in> ListItem(V:K) => . ...</in>
```

In the case studies, we show how this mechanism can be put into practice.

6.2.2 State transitions

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

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

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

6.2.3 LTL

Before showing how we augmented $\mathbb{KPHP}$ to handle the verification of LTL properties, we provide an informal introduction to the topic of temporal logics and Linear Temporal Logic (LTL) in particular. Quoting [120], "*temporal logic extends propositional or predicate logic by* modalities *that permit to refer to the infinite behaviour of a reactive system*".

In general, a temporal logic is obtained by augmenting a propositional logic with a set of *temporal connectives* such as $\diamond$ ("eventually"), $\Box$ ("always"), X ("next") and U ("until"), allowing the expression of properties such as

- $\diamond P$: "eventually (i.e. in some future state), the property P becomes true".
- $\Box P$: "property P is true, and will be true in all subsequent states".
- $P_1 U P_2$: "property P_1 is true, and will be true until P_2 becomes true".
- XP : "property P will be true at the next step".

In temporal logics, time can be either *linear* or *branching*. When time is linear, given a point in time (i.e. a state of the system) there is only one successor. On the other hand, when a notion of branching time is used, there could be many successors, expressing the notion of *possibility*. As implied by its name, Linear Temporal Logic (LTL) is defined over a linear time notion. In their basic forms, temporal logics refer to a *discrete time*, where, intuitively, the present moment is seen as the current state and the next moment represent the next state. In other words, discrete time allows the specification of the relative order of events, but not the actual moments in physical time in which they occur [120].

Syntax of LTL

The set of LTL formulae can be defined as follows

$$\phi ::= true \mid false \mid a \in AP \mid \phi_1 \wedge \phi_1 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 \rightarrow \phi_2 \mid \diamond \phi \mid \Box\phi \mid \phi_1 U \phi_2 \mid X\phi$$

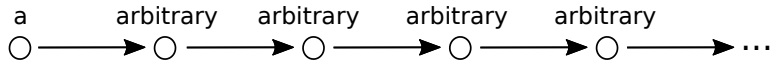
where AP is a set of *atomic propositions* defined in the context of application. For example, in our temporal logics for PHP, we define atomic propositions such as "variable `$x` contains the value `5`" or "variables `$x` and `$y` are aliased". We describe our atomic propositions in great detail later in this section.

Semantics of LTL

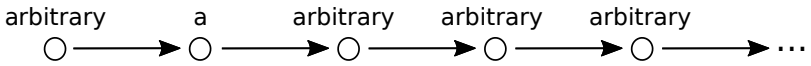
We informally describe the semantics of LTL formulae, in order to convey the reader the set of operational intuitions necessary to understand our contribution. A more detailed and formal

treatment of the topic can be found in [120].

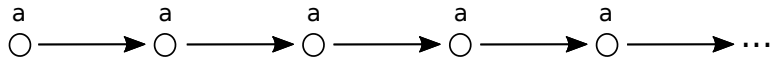
Let us start by considering an atomic proposition $a \in AP$. Then, the LTL formula $\phi = a$ can be simply interpreted as "the proposition a is true in the current state", as shown by the following diagram



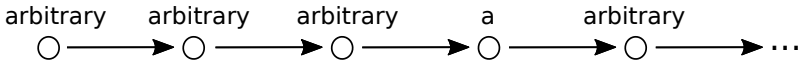
Notice that no constraints are expressed about the future states, in which a can be either true or false. Consider now the use of "next" as in the formula $\phi = X\ a$. This expresses the fact that, *at the next discrete time step*, a must be true:



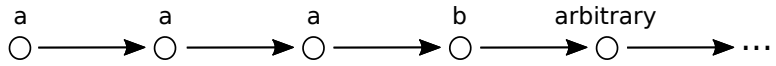
Consider instead the formula $\phi = \Box\ a$, which uses the "always" LTL connective. This formula express that the proposition a must be true at the current time/state, and also in all subsequent ones:



The "eventually" connective instead expresses the fact that the property will eventually become true at some point in the evolution of the system. The intuitive meaning of $\phi = \diamond\ a$ is depicted in the following diagram:



Finally, given another atomic proposition $b \in AP$, the formula $\phi = a\ U\ b$ expresses the fact that a is true *until* b finally becomes true at some point in the future, as expressed by the diagram:



Examples of LTL formulas

To clarify the concept introduced above, we now show a couple of classical examples of applications of Linear Temporal Logic. More detailed examples applied to the verification of PHP code will follow.

Mutual exclusion. Consider two concurrent processes P_1 and P_2. Each can be thought as having a *non-critical* section, a *waiting phase* and a *critical section*. Let us define two propositions $critical_1$ and $critical_2$, denoting the fact that P_1 (respectively P_2) is in its critical section. A *safety property* expressing the fact that, at each point in time, P_1 and P_2 never access their critical section simultaneously can be expressed by the simple LTL formula

$$\Box(\neg critical_1 \vee \neg critical_2)$$

Conversely, a *liveness* property stating that each process will be able to infinitely often enter its critical section, i.e. no process would get stuck, is expressed by the LTL formula

$$(\Box \diamond critical_1) \wedge (\Box \diamond critical_2)$$

Traffic light. Consider a model of a traffic light having three states "green", "yellow" and "red". The formula

$$\Box \diamond green$$

encodes the liveness property "the traffic light is infinitely often green", needed to avoid the traffic light causing the entire traffic to get stuck for an undefined amount of time. LTL formulae can also be used to specify the expected behaviour of the traffic light. Consider for example the specification "once red, the light can't become immediately green". This is expressed by the LTL formula

$$\Box(red \rightarrow \neg X\ green)$$

In many countries indeed, after being red, the traffic light becomes yellow for a while before finally becoming green. This can be specified by the LTL formula

$$\Box(red \rightarrow X(red\ U\ (yellow \wedge (yellow\ U\ green))))$$

6.2.4 A temporal logic for PHP

We now define an extension of LTL with predicates over $\mathbb{KPHP}$ configurations (i.e. PHP program states). Given a PHP program, we would like to be able to express conditions such as: "variable `$usr` never contains `'admin'`", or "the local variable `$y` of function `foo` will at some point be aliased to the global variable `$y`". The LTL formulas corresponding to the informal specifications above are, respectively:

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

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

To this end, we introduce predicates such as `eqTo` (equals to) and `alias`, where `var('x')` denotes a variable name `'x'`, `val(v)` a value `v`, `gv` a global variable and `fv('fun',var('x'))` a local variable (of a specific function `fun`).

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

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

meaning, that formula `eqTo(e1,e2)` is true for `B` if and only if `e1` and `e2` evaluate to the same value. The definition of functions such as `eval` is crucial, as it connects the semantics to the model checker. These functions should be written in purely functional style and avoid side effects. In practice they are pretty simple, as they only need to inspect a configuration using pattern matching. For example, the function `eval` may take a value or a variable (either global or local) as input; if given a value it simply returns the value itself, unmodified. If given a variable name it reads it

from the configuration and returns its value. As another example, consider the `alias(e1,e2)`:

$$\texttt{B} \models \texttt{alias(e1,e2)} \Leftrightarrow \texttt{lvalue(B,e1) = lvalue(B,e2)}$$

The function `lvalue` returns the location of the heap where its argument is stored, so the predicate is true when the arguments are aliased or identical.

Moreover, we also want to be able to reason about *correspondence assertions* [121], by labelling program points and stating properties such as "after reaching label `'login'` variable `$sec` always contains 1". Formally:

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

In order to support the use of labels we instrumented the semantics by including a `<label>` cell to the configuration together with an auxiliary construct `label` (available to the programmer as a function) which, when evaluated, simply updates the current label (i.e. the value contained in the `<label>` cell). The predicate `label('foo')` is true in any configuration where the current label is `foo`:

$$\texttt{B} \models \texttt{label(X)} \Leftrightarrow \texttt{get\_config\_label(B) = X}$$

where the function `get_config_label` simply return the value stored in the `<label>` cell.

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

$$\Box(\texttt{inFun('foo')} \Rightarrow \texttt{has\_type(fv('foo',var('y')),integer)})$$

The predicate `has_type` matches the type of the first argument (after its evaluation) against the type given as second argument

$$\texttt{B} \models \texttt{has\_type(E,T)} \Leftrightarrow \texttt{type(eval(E)) = T}$$

where the function `type` is defined by simple pattern matching on the possible types of its input value. The predicate `inFun(X)` is true in any configuration where the function being executed is `X`

$$\texttt{B} \models \texttt{in\_fun(X)} \Leftrightarrow \texttt{get\_current\_fun(B) = X}$$

similarly to the previous cases, the function `get_current_fun` simply retrieves the name of the function which is currently being executed (by inspecting the top-most stack frame). If no function is being executed, it returns the value `'top-level'` instead.

We use similar techniques to give semantics to all of our predicates, which can be used to form extended LTL formulas together with standard LTL connectives. A summary of our predicates together with their definition and some examples is provided in Table 6.1.

Predicate	Definition	Example
eqTo(e1,e2)	eval(B,e1) = eval(B,e2)	□¬eqTo(gv(var('usr')),val('admin'))
geq(e1,e2)	eval(B,e1) ⩾ eval(B,e2)	◇ geq(len(gv('pswd')),val(8))
leq(e1,e2)	eval(B,e1) < eval(B,e2)	leq(len(gv('urs')),gv('max'))
alias(x,y)	lvalue(B,e1) = lvalue(B,e2)	◇alias(fv('foo',var('y')),gv(var('y')))
label(X)	get_config_label(B) = X	label('login')⇒ □eqTo(gv(var(sec)),val(1))
in_fun(F)	get_current_fun(B) = X	inFun('foo')⇒ eqTo(gv('count'),val(1))
has_type(E,T)	type(eval(E)) = T	has_type(fv('foo',var('y')),integer)

Table 6.1: Summary of the LTL atomic predicates defined on $\mathbb{KPHP}$ configurations.

6.2.5 Example

Before examining the real-world case studies of the next Section, let us consider the following introductory example

```
$y = 0;
function foo() {
        $z = "test";
        global $y;
        $x = &$y;
}
foo();
```

where, in the body of function `foo`, the global variable `$y` is aliased to the local variable `$x` (notice the use of the `global` keyword to access `$y`); also, function `foo` defines an additional local variable `$z` containing the `string "test"`. Using LTL augmented with our predicates, we can easily state

the following simple properties, and prove them true by invoking $\mathbb{K}$/Maude's builtin model checker:

1. "Global variable `$y` and local variable `$x` will be aliased at some point during program execution":

$$\Diamond\texttt{alias(gv(variable("y")),fv("foo", variable("x")))}$$

2. "Variable `$y` will never be assigned values of type `string`":

$$\neg\Diamond\texttt{hasType(gv(variable("y")),string)}$$

3. "Global variable `$z` is *never initialised* (i.e. it always evaluates to `NULL`)":

$$\Box\texttt{eqTo(gv(variable("z")),val(NULL))}$$

Counterexamples

Until now, we have been focusing on proving that a given property *holds*. What if instead the property is false? In this case, the model checker will display a *counterexample* - an execution trace (a sequence of standard $\mathbb{KPHP}$ configurations) for which the property is false.

For example, suppose we want to prove that, in the program above, "(global) variable `$y` is never initialised":

$$\Box\texttt{eqTo(gv(variable("y")),val(NULL))}$$

Intuitively, the formula states that "for every program state encountered during execution, variable `$y` must evaluate to `NULL`", but this is clearly false, since `$y` is assigned the value `0`. This causes the model-checker to output an execution trace (as a sequence of $\mathbb{KPHP}$ configurations)

$$c_1 \to c_2 \to \cdots \to c_n$$

for which the property is false. In the example, a quick inspection of the configurations in the

trace shows that after its initial assignment, `$y` contains the value `0`, disproving the claim that `$y` is never initialised.

6.3 Case studies

In this section we show how we used LTL model checking in conjunction with symbolic execution to analyse properties of two 3rd-party PHP functions of practical relevance. The discussed case studies (alongside other examples) are included in the ℍℙℍℙ distribution. Moreover, the ℍℙℍℙ web interface (`http://phpsemantics.org`) exposes the model checking and symbolic execution functionalities discussed in this chapter and makes it possible to input PHP code together with symbolic input and an LTL formula to be verified. The case studies are included as examples in the web interface and could be loaded and run from there (without having to download the whole ℍℙℍℙ distribution).

6.3.1 Input Validation

In our first example of model checking, we consider the function `PMA_isValid` taken from the source code of `phpMyAdmin` [122], one of the most common open source web applications, which provides a web interface to administer an SQL server.

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

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

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

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

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

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

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

Hence, `PMA_isValid` is a "swiss knife" function used to actually compare variable values, or their type, or to check wether they have a positive string length or belong to a given array, where the intended behaviour/execution path is selected as a result of a combination of the values and types of the three arguments.

The developer had an informal specification of this function in mind, which he wrote in a comment at the beginning of the function:

```
/**
 * checks given $var against $type or $compare
 *
 * $type can be:
 * - false       : no type checking
 * - 'scalar'    : whether type of $var is integer, float, string or boolean
 * - 'numeric'   : whether type of $var is any number repesentation
 * - 'length'    : whether type of $var is scalar with a string length > 0
 * - 'similar'   : whether type of $var is similar to type of $compare
 * - 'equal'     : whether type of $var is identical to type of $compare
 * - 'identical' : whether $var is identical to $compare, not only the type!
```

```
 * - or any other valid PHP variable type
 **/
```

However, it is not obvious wether such specification is met by the actual implementation. Leveraging model checking and symbolic execution we are able to prove that the function behaves as expected, by verifying each sub-case.

In order to do that, we first write some code accepting (possibly) symbolic inputs and calling the function:

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

then we attempt to verify multiple times the LTL formula

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

each time providing different combinations of symbolic and concrete inputs (by initialising the input stream with the appropriate values), until all of the cases discussed above are covered. Indeed, these verifications succeed, proving the correctness of the function.

As a concrete example, in order to prove that the result of calling `PMA_isValid` with `$type="numeric"` is `true` when `$var` is an `integer`, we provide the symbolic input `#symInt(x)` to `$var`, and the concrete input `"numeric"` to `$type`. We proved analogous results for the case of `float` variables, and for the other similar cases. We proved that `PMA_is_Valid($var, "similar", $compare)` returns `true` for any `integer $var` and `string $compare`, by providing symbolic values `#symInt(x)` and `#symString(y)` to `$var` and `$compare`. Each of the queries took about $30s$ to be verified by the model checker.

6.3.2 Cryptographic Key Generation

In our second example, we consider the Password-Based Key Derivation Function `pbkdf2` from the PHP distribution [123]. `pbkdf2` takes five parameters: the name of the algorithm to be used for hashing (`$algo`), a `$password`, a `$salt`, an iteration `$count` and the desired `$key_length`. It returns a key derived from `$password` and `$salt` whose length is `$key_length`. We wish to prove that the function always returns a `string`, and that its length is equal to the requested `$key_length`.

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

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

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

1. The result is a string: $\Diamond$`has_type(gv(var('result')),string)`

2. The length of the output is as requested:

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

   where the function `len` simply returns the length of the input string.

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

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

Property (3) shows that property (2) is non-trivial. Moreover, it illustrates a technically more intriguing LTL formula involving a great number of atomic predicates and temporal connectives. The formula states that, at each point of execution (encoded by the top-level connective $\Box$), the following must be true: if the current function is `pbkdf2` and control will eventually return to the top-level, then the length of the string stored in the local variable `$output` must eventually become greater than the value stored in the local variable `$key_len`, before the function returns. As already mentioned, this shows that property (2) is non-trivial; in fact, since the output calculated by `pbkdf2` eventually exceeds the required length, in order for (2) to be satisfied the output necessarily have to be trimmed/processed before the function returns.

Using the model checker, we are able to verify all three properties in less than one minute. Unlike the previous example, which we were able to run and analyse out-of-the-box, in this case we had to provide implementations for an number of functions (such as `hash`), which belong to libraries outside of the core language. For the sake of verification, we only provide simple stubs for these functions, making sure to preserve the type and output length properties of their original versions. The complete source code of `pbkdf2` and related functions can be found in Appendix A.2.

6.4 Discussion and Limitations

The approach described here suffers from the known limitations of the underlying verification techniques. In particular, explicit state LTL model checking will struggle to handle programs that generate large state spaces that depend heavily on the program inputs. The support for symbolic execution mitigates this problem, but as common to this approach it does not handle higher order data structures, such as objects, and struggles with loops depending on symbolic values. However, despite these limitations, we have shown that we can indeed verify some non-trivial properties of real PHP code, by leveraging the existing support for LTL model checking and symbolic execution provided by the $\mathbb{K}$ framework.

In the next Chapter, we develop a more sophisticated static analyser for PHP, based on the

theory of Abstract Interpretation and also derived from our formal semantics. In particular, our static analyser is modular and extensible, allowing users to define and perform their own static analysis, under a unified framework.

Chapter 7

A general purpose static analyser for PHP

Armed with $\mathbb{KPHP}$, our formal semantics of PHP, together with the conceptual framework provided by the theory of abstract interpretation (introduced in 4.5), we now detail the development of a sound-by-construction, modular, general purpose and extensible static analyser for PHP, which we call $\mathbb{KPHP}^{\#}$.

The final outcome is an executable formal semantics, also written in $\mathbb{K}$, which closely resembles the original $\mathbb{KPHP}$ but whose underlying domain of computation (referred to as abstract domain in the following) is a parameter. The chosen abstract domain, which can be seen as a *plugin* to the system, determines the analysis. For example, we have already defined abstract domains for computing signs (useful e.g. for detecting division-by-zero) and types (in order to perform type-checking and detect possibly dangerous or dubious constructs) while we are planning to define a new domain for taint-checking (which can be used to detect taint-style vulnerabilities such as XSS, SQLi and command injection, as described in section 3.3), among others. The main idea is then to provide a single unified framework which takes care of performing most of the analysis, independently of the chosen domain, so that the chosen specific analysis can then be performed on top of it, in a modular way.

In order to execute a program (as opposed to analysing it), one does not need to switch to the original $\mathbb{KPHP}$. In fact, $\mathbb{KPHP}^{\#}$ is able to both execute and analyse programs, under

a unified framework. Executing a program in $\mathbb{KPHP}^{\#}$ amounts to "analysing" it in a particular *concrete* domain whose values are the usual PHP data types and the abstraction and concretisation functions are the identity:

$$\alpha(x) = x = \gamma(x)$$

7.1 Methodology and overview

Before detailing the development of $\mathbb{KPHP}^{\#}$ we first provide the intuition behind it. Given the initial $\mathbb{KPHP}$, and the set of PHP values $\mathcal{V}_{PHP}$ we obtain our general purpose static analyser in the following way:

Defining an abstract domain. As a first step, we define an abstract domain $\mathcal{A}$ together with an abstraction function $\alpha : \mathcal{V}_{PHP} \to \mathcal{A}$. As mentioned before, this could also be the identity function, specifying a "concrete" abstract domain, with no loss of information. In general, however, an abstract domain specifies the domain of an analysis, such as signs, types, taintedness, etc.

Abstracting values. The next step is to update $\mathbb{KPHP}$ in such a way that literals (as found the source code) are evaluated according to the chosen domain, as specified by the abstraction function α. In other words, we need to specify how abstraction is introduced in the system. For example, the literal `3` in the fragment `$x = 3;` would be abstracted into $\alpha($`3`$)$. This is accomplished by simply modifying the existing rules that take care of evaluating literals returned by the parsers into values, as well as those that mention constant literals directly.

Identifying the domain operations. Once values have been abstracted according to the chosen domain, the existing operations on values will not be valid anymore. For example, once numeric values have been abstracted, say, to their sign, the usual arithmetic operations will have to be redefined. Therefore, our next step is to identify all parts of the semantics that explicitly depend on domain elements, and factor them out into a separate module. This includes the obvious arithmetic operations but also rules for type conversions, output and more. Once identified, this

constitutes a set of operations that need to be defined *for each* domain, and constitute, together with the definition of the domain itself, a plugin to the system. Notice that, at this stage, it is already possible to abstractly execute simple programs that do not contain control structures.

Lifting the semantics. The final step consists in lifting the semantics so that handling of multiple execution branches becomes possible in the presence of abstract values. This includes conditionals, loops, function calls as well as other control flow operations such as `break` and `continue`. Consider for example a conditional statement

```
if ($x == 2)
        $y = 1;
else
        $y = -1;
```

and suppose that variable `$x` evaluates to an abstract value such as, say, one denoting a positive integer (as seen in 4.4). The single abstract value to which `$x` evaluates is therefore a machine encoding of a *set* of concrete values, namely the set of all positive integers in this case. Clearly, the value `2`, included in the set, causes the conditional above to take the true branch, while all other values cause it to take the false branch. However, since the actual concrete value is not known (i.e. all is known is that `$x` evaluated to an abstract value denoting a positive integer), both branches need to be considered, and the different outcomes need to be combined in a sound manner.

We imposed a constraint on our development, that the abstract version of the rules should be a proper generalisation of the concrete version, meaning that, when switching to the identity abstract domain (where $\alpha(x) = x$), the same rules should provide concrete execution. We develop such abstract rules for conditional, loops, function call, array/object access etc. following the principles of abstract interpretation (as found in [94]). Moreover, we also abstract in a similar way a number of concrete rules involving side conditions. Consider for example the following idealised $\mathbb{K}$ rule

```
rule doSomething(X) => 1 when isInteger(X)
rule doSomething(X) => 2 when notBool isInteger(X)
```

When switching to abstract domains, rules such as this one must also be updated in order to take into account that the argument's type may be unknown and therefore the exact truth value of the `isInteger` predicate may not be available. To deal with this situation correctly, both cases must be considered independently; as for conditionals, the obtained results must then be combined into one.

7.2 Parametrising the domain of computation

In this section we show how we separate domain operations from non-domain ones in $\mathbb{KPHP}$ and how we take advantage of this separation in order to allow arbitrary domains to be defined and used. As explained in 4.5, the linking between concrete domain (the set of usual PHP values) and abstract domain is given by the abstraction function α of the given abstract domain.

In the following, we shall work through a simple running example, the abstract domain of *signs* (as introduced in 4.5.2), however our reasoning and techniques are general enough to be applied to other abstract domains. In particular, we show how, by simply factoring out a relatively small set of rules from $\mathbb{KPHP}$ and by applying the abstraction function α, we are already able to obtain a usable abstract interpreter which can correctly analyse simple arithmetic programs. In the next section we show how to further abstract the semantics so that also programs containing conditionals, loops, recursive function calls, complex array or object manipulation could be correctly analysed.

To keep our development better organised, we thus split the main $\mathbb{KPHP}$ source folder into two subfolders: `lang` and `domains`, as shown if Figure 7.1. The `lang` folder contains the set of rules that are common to *all* domains i.e. whose definition is independent from the particular abstract domain. The `domains` folder will in turn contain a set of folders, one for each abstract domain, each containing the set of rules that had to be redefined for that particular domain (such as arithmetic operations, type conversions, and, in general, every other operation which *directly* manipulates values). At each time, only one domain may be chosen and imported as a module via a standard $\mathbb{K}$ import command. To change the abstract domain and obtain a different analyser, it suffices to

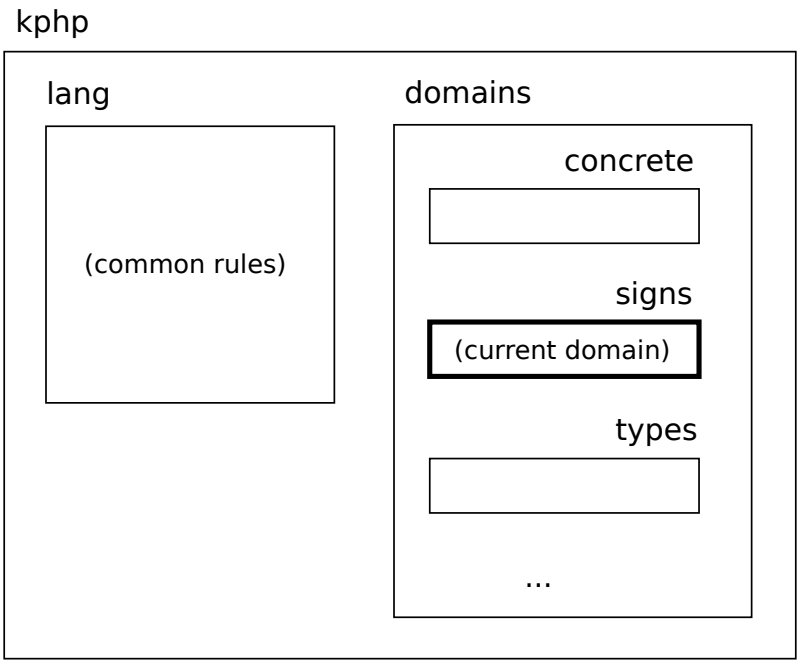


Figure 7.1: The folder structure of $\mathbb{KPHP}^{\#}$. Common rules are separated from domain-dependent rules. Many domains can be available, but only one (highlighted) may be chosen and imported in the semantics. To obtain a different analyser, the semantics must be re-compiled importing a different domain.

import a different module and recompile the system.

7.2.1 Abstract domains

Recall that in PHP we have scalar (`int`, `float`, `bool` and `string`) and compound types (`array` and `object`), and that compound objects, say arrays, are defined in terms of scalar values, as mappings from keys (i.e. `int` or `string`) to locations (which in turn contain other values). In our semantics rules in `memory.k`, this looks like

```
syntax CompoundValue ::= Object | Array
syntax ScalarValue ::= Int | Float | Bool | String
```

where `Object` and `Arrays` are defined as described in 5.2.1 and `Int`, `Float` etc. are $\mathbb{K}$'s builtin types. We modify the above definition of scalar values by instead using *abstract integers* (`AbstractInt`),

abstract booleans (`AbstractBool`) and so on:

```
syntax CompoundValue ::= Object | Array
syntax ScalarValue ::= AbstractInt | AbstractFloat | AbstractBool | AbstractString
```

The new abstract scalar values will be defined in a separate module and may be different in different abstract domains. Note also that by simply parametrising scalar values, objects and arrays (and every other construct defined in the semantics) will still be defined as before and no change is required (except for the trivial update of the involved types so that the generalised abstract ones are used instead). For example, arrays will now be maps from abstract integers or abstract strings to locations, however abstract strings and integers are defined in the currently loaded domain.

In our framework, it is important to preserve the existing structure of scalar values in order to facilitate the translation of concrete rules into abstract ones while maintaining modularity. Consider a hypothetical (concrete) rule R. If R contains an explicit match against a variable of a scalar type, this would necessarily be of type `int`, `float`, `bool` or `string`. To transform the concrete rule into an abstract one, we need to replace all scalar types with their abstract counterparts `AbstractInt`, `AbstractFloat` etc. before finally updating the rest of the rule accordingly by replacing every elementary operation with its abstract version. Consider the (concrete) case of addition

```
'Plus(L:Int, R:Int) => L +Int R
'Plus(L:Float, R:Float) => L +Float R
```

where `+Int` and `+Float` are $\mathbb{K}$'s builtin addition operators for integers and floats respectively. After switching to the abstract domain, the rules above are easily transformed into

```
'Plus(L:AbstractInt, R:AbstractInt) => L +AbstractInt R
'Plus(L:AbstractFloat, R:AbstractFloat) => L +AbstractFloat R
```

where `+AbstractInt` and `+AbstractFloat` are now custom-defined operations for the given abstract integers and floats (we discuss them in detail later in this Section). An important fact to notice is that the new abstract rules are valid for *every* choice of `AbstractInt` and `AbstractFloat` - no matter how (abstract) integers and floats are defined in a particular domain, as long as the `+AbstractInt` and `+AbstractFloats` operations are provided, the rules for `'Plus` will still be valid and therefore

rewriting them for each domain is not required (instead, a single, factored-out version could be used).

On the other hand, suppose that we employed a more arbitrary abstraction such as

```
syntax CompoundValue ::= Object | Array
syntax ScalarValue ::= T1 | T2 | ... | Tn
```

where we have defined n arbitrary "abstract scalar types". How can we translate the concrete rules for `'Plus` into abstract ones? Without the subdivision of scalars inherited by the concrete domain this is not immediately obvious. In the worst case, we could imagine ourselves defining n ad-hoc rules for `'Plus`, one for each `T1`, `T2` etc.:

```
'Plus(L:T1, R:T1) => L +T1 R
'Plus(L:T2, R:T2) => L +T2 R
...
```

Since, in this scenario, each domain is allowed to define arbitrary scalars (instead as being constrained to abstract integers, floats, booleans and strings), those rules are not general as in the previous case. Instead, they are only valid for the given domain, and each new domain would have to define its own rules for `'Plus` as well as the rules for addition on the defined types (`+T1`, `+T2` etc.).

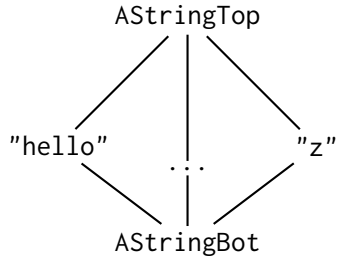
Clearly, this strategy is not only error-prone, but it also breaks our goal of developing a modular framework where new abstract domains could be plugged in with minimal effort without having to rewrite most of the semantics. This strategy, when taken to the extremes, entails rewriting/updating the whole semantics for every domain, leading us back to a special-purpose analyser, which is the opposite of our desired outcome, a general-purpose and extensible static analyser. We therefore trade some flexibility in exchange for greater modularity, extensibility and ease of use from the point of view of the developer interested in developing their own analysis.

The next step consists in providing an actual definition for abstract values. This consists of a lattice, encoding the domain of computation, together with an abstraction function α mapping concrete scalar values to abstract ones (see 4.5.2). We write such definition in a file `lattice.k`,

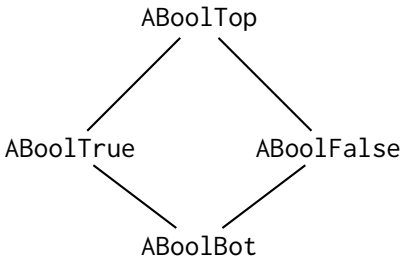
located inside a particular domain's folder. The following fragment of `lattice.k` shows how we can define abstract scalar values for a simple sign domain which abstract numeric values into their signs (keeping track of the different types of integer and floats), booleans into abstract booleans (similar to abstract integers but of different type), and keep strings concrete:

```
module LATTICE
...
syntax  AInt ::= "AIntTop" | "AIntBot" | "AIntZero" | "AIntPos" | "AIntNeg"
syntax  AFloat ::= "AFloatTop" | "AFloatBot" | "AFloatZero" | "AFloatPos" | "AFloatNeg"
syntax  ABool ::= "ABoolTrue" | "ABoolFalse" | "ABoolTop" | "ABoolBot"
syntax  AString ::= "AStringTop" | String | "AStringBot"
...
```

In particular, the intended lattice structure for numeric values is the same we introduced in 4.5.2, while, for strings, it looks like the following,



essentially representing the set of concrete strings augmented with the top and bottom elements of the lattice (notice that therefore the lattice is conceptually infinite). Similarly, for booleans we model the lattice



which encodes the usual booleans lifted with the top and bottom elements. In particular, the abstract boolean top represents an unknown truth value.

Once the elements of the abstract domain are defined, we need to provide definitions for the usual lattice operations such as the *least upper bound* and the *lattice ordering*. For this simple domain, the least upper bound for numeric values is simply defined as a set of rules encoding the table shown in 4.5.2:

```
syntax  AInt ::= "lub_AInt" "(" AInt "," AInt ")" [function]
// ---
rule    lub_AInt(AIntBot,X) => X
// ---
rule    lub_AInt(AIntNeg,AIntBot) => AIntNeg
rule    lub_AInt(AIntNeg,AIntNeg) => AIntNeg
rule    lub_AInt(AIntNeg,AIntZero) => AIntTop
rule    lub_AInt(AIntNeg,AIntPos) => AIntTop
rule    lub_AInt(AIntNeg,AIntTop) => AIntTop
// ---
rule    lub_AInt(AIntZero,AIntBot) => AIntZero
rule    lub_AInt(AIntZero,AIntNeg) => AIntTop
rule    lub_AInt(AIntZero,AIntZero) => AIntZero
rule    lub_AInt(AIntZero,AIntPos) => AIntTop
rule    lub_AInt(AIntZero,AIntTop) => AIntTop
// ... more rules here ...
```

(the float case is analogous), while for strings we simply define

```
syntax  AString ::= "lub_AString" "(" AString "," AString ")" [function]
// ... Top and Bottom cases here ...
rule    lub_AString(S1:String,S2:String) => S1 when (S1 ==String S2)
rule    lub_AString(S1:String,S2:String) => AStringTop when (S1 =/=String S2)
```

encoding the fact that the least upper bound of two concrete strings is Top if the two strings are different (representing an uncertain value) or the string itself if they are the equal. Boolean are treated in a similar fashion, by keeping their concrete value when possible and by using `ABoolTop` when uncertain. Importantly, for booleans we also have to define, as part of every domain, a pair

of predicates $isValid$ and $isUnsat$

$$isValid : ABool \to Bool$$

$$isUnsat : ABool \to Bool$$

which we will use later to handle control flow in the analysis. For our simple sign domain, we define

```
syntax  Bool ::= "isValid" "(" ABool ")" [predicate]
rule    isValid(B) => B ==K ABoolTrue
// ---
syntax  Bool ::= "isUnsat" "(" ABool ")" [predicate]
rule    isUnsat(B) => B ==K ABoolFalse
```

The meaning and usage of those predicates will become clear in the next section where we show how to abstract control-flow operations. For now, note that we defined `ABoolTrue` as valid and `ABoolFalse` as unsatisfiable, encoding the fact that, in this domain, `ABoolTrue` and `ABoolFalse` are respectively "definitely true" and "definitely false". The remaining two elements of the lattice, the top and bottom elements in this case, are neither valid nor unsatisfiable, encoding an uncertain truth value. The lattice ordering is also defined as a set of rules encoding the ordering of the chosen lattice:

```
syntax  Bool ::= K "<Lattice" K         [function]
// ---
rule    AIntBot <Lattice AIntBot => false
rule    AIntBot <Lattice X => true when X =/=K AIntBot
// ---
rule    AIntTop <Lattice X => false
// ---
rule    AIntNeg <Lattice AIntBot => false
rule    AIntNeg <Lattice AIntNeg => false
rule    AIntNeg <Lattice AIntZero => false
rule    AIntNeg <Lattice AIntPos => false
```

```
rule    AIntNeg <Lattice AIntTop => true
// ... more rules here
```

Notice that we are able to provide such definitions in a simple tabular form in this case, but more clever techniques may be required in other domains. In particular, we recommend implementing a general-purpose library for dealing with lattices in $\mathbb{K}$ (as a $\mathbb{K}$ `MODULE` which could be imported in any $\mathbb{K}$ definition).

7.2.2 Abstraction

Once the abstract domain is defined, it has to be put into use by providing an abstraction function to map PHP values into abstract values. For each abstract domain, we define the abstraction function in the `alpha.k` file. For the sign domain example, the abstraction function is slightly more than a $\mathbb{K}$ encoding of the function discussed in 4.5.2:

```
module ALPHA
imports LATTICE

syntax K ::= "alpha" "(" K ")" [function]
// ---
// abstraction of integers
rule  alpha(N:Int) =>
  #if (N ==Int 0) #then
    AIntZero
  #else
    #if (N >Int 0) #then
      AIntPos
    #else
      AIntNeg
    #fi
  #fi
// ---
// abstraction of floats
```

```
rule  alpha(F:float) =>
  #if (N ==Float 0.0) #then
    AFloatZero
  #else
    #if (N >Float 0.0) #then
      AFloatPos
    #else
      AfloatNeg
    #fi
  #fi
// ---
// "abstraction" of strings
rule    alpha(S:String) => S
// ---
// abstraction of booleans
rule  alpha(true) => ABoolTrue
rule  alpha(false) => ABoolFalse
```

The above rules simply map numeric values to the abstract value (as defined in `lattice.k`, required by the module) encoding their sign, and strings and booleans to their equivalent representation in the domain.

7.2.3 Putting the domain into action

Given our domain of signs and an abstraction function α mapping PHP values to abstract values, we need to update $\mathbb{KPHP}$ so that it makes use of the new domain. This consists in updating the rules that evaluate AST nodes for literals (defined in `core.k`) so that, instead of simply returning the literal encoded in the AST node as returned by the parser, they first apply the abstraction function α to it. The following snippet of code shows the updated $\mathbb{K}$ rules, where the only change we applied is the use of the abstraction function `alpha`:

```
rule [literal-int]:
  <k> 'LNumber('Deci(Str:String)) => alpha(String2Int(Str)) ... </k>
```

```
  [structural]
// ---
rule  [literal-float]:
  <k> 'DNumber(Str:String) => alpha(String2Float(Str)) ... </k>
  [structural]
// ---
rule [literal-string]:
  <k> 'Literal(Str:String) => alpha(Str)      ... </k>
  [structural]
// ---
rule [literal-true]:
  <k> 'True(_) => alpha(true) ... </k>
  [structural]
// ---
rule [literal-false]:
  <k> 'False(_) => alpha(false) ... </k>
  [structural]
// ---
rule [literal-null]:
  <k> 'Null(_) => alpha(NULL) ... </k>
  [structural]
```

Once the structure is in place, the desired domain could be imported via `core.k` as in the following snippet

```
module CORE
  imports SHARED
  imports PROGRAM-TRANSFORMATIONS
  imports BUILTIN-FUNCTIONS
  // chose a domain!
  imports SIGNS-DOMAIN
  // ...
```

However, as mentioned before, the existing domain operations such as arithmetic expressions would

not work anymore in this settings, since they were based on the $\mathbb{K}$ builtin types while now we have switched to arbitrary domains (each defining their own data types). The next step is therefore to identify which semantic rules of $\mathbb{KPHP}$ explicitly operate on values and isolate them so that they will instead be part of the specific domain.

The reader may notice that although we've been discussing how to abstract language values (*data abstraction*) via an abstract domain of choice, we haven't mentioned how to deal with the abstraction of the memory itself (*heap abstraction*). The reason for this is that while the data abstraction can be freely selected (or created from scratch) by the user, the heap abstraction is currently built-in and cannot be replaced; in particular, we base our approach on the common *non-relational* abstraction as found in [94] and discussed in Section 4.5. By building on a solid, general purpose heap abstraction we allow our users to focus on developing their own abstract domains for data easily and without worrying too much about more involved details. In future versions of the framework, we may consider parametrising heap analysis and allowing advanced users users to chose between different options or developing their own.

7.2.4 Partitioning the semantics by isolating domain operations

In order to obtain a first working abstract interpreter (in this example, to analyse the sign of variables), we need to provide appropriate definitions for arithmetic operations as well as other value-manipulating operations. To do that, we create a new file `operations.k` for every domain, to be placed in the domain folder. We then identify the $\mathbb{KPHP}$ rules which need to be redefined and we move them into this file.

For arithmetic and logical operations (addition, multiplication, division, equality etc.), this is simple as all we have to do is (similarly to what was already done for the least upper bound relation) to translate into $\mathbb{K}$ rules the abstract arithmetic operations we have already derived in tabular form in 4.5.2:

```
// Plus
syntax  AInt ::= AInt "+AInt" AInt [function]
```

```
// ---
rule    AIntBot +AInt X => AIntBot
// ---
rule    AIntNeg +AInt AIntBot => AIntBot
rule    AIntNeg +AInt AIntNeg => AIntNeg
rule    AIntNeg +AInt AIntZero => AIntNeg
rule    AIntNeg +AInt AIntPos => AIntTop
rule    AIntNeg +AInt AIntTop => AIntTop
// ---
rule    AIntZero +AInt AIntBot => AIntBot
rule    AIntZero +AInt AIntNeg => AIntNeg
rule    AIntZero +AInt AIntZero => AIntZero
rule    AIntZero +AInt AIntPos => AIntPos
rule    AIntZero +AInt AIntTop => AIntTop
// ...
```

However, while in a simple textbook example this is enough, in $\mathbb{KPHP}$ there are other operations that directly manipulate values and therefore need to be redefined as well, for each domain. One example is the type conversion rules (pervasive in PHP, see 3.2). The abstract version of those rules should provide a sound approximation of the behaviour specified by the concrete rules themselves. Consider for example a fragment of the abstract version of the `toFloat` rule, which takes as input any PHP type and converts it into a `float`:

```
syntax  K ::= "*toFloat" "(" LanguageValue ")"  [strict]
// ---
rule [toFloat-BFalse]:
  <k> *toFloat(BFalse) => alpha(0.0) ... </k>
  [internal]
rule [toFloat-BTrue]:
  <k> *toFloat(BTrue) => alpha(1.0) ... </k>
  [internal]
rule [toFloat-BTop]:
  <k> *toFloat(BTop) => AFloatTop ... </k>
```

```
  [internal]
// ---
rule [toFloat-AInt]:
  <k> *toFloat(I:AInt) => AFloatTop ... </k>
  [internal]
// ---
rule [toFloat-AFloat]:
  <k> *toFloat(F:AFloat) => F ... </k>
  [internal]
// ---
rule [toFloat-AString-concrete]:
  <k> *toFloat(S:String) => *toFloat(string2Number(S)) ... </k>
  [internal]
rule [toFloat-AStringTop]:
  <k> *toFloat(AStringTop) => AFloatTop ... </k>
  [internal]
// ---
rule [toFloat-compound]:
  <k> *toFloat(C:CompoundValue) => WARNING("conversion of compound
  types to float is undefined
  (http://www.php.net/manual/en/language.types.integer.php)\n")
  ~> alpha(1.0) ... </k>
  [internal]
```

The rule closely resembles its concrete counterpart, but it has been adapted to take into account the presence of abstract, uncertain values. For `true` or `false` booleans, the abstract counterpart of the expected concrete value is returned. For integers, we conservatively return `AFloatTop` (but notice that more precise decisions could be made by performing a case analysis on the input integer value). The cases for float, string and compound inputs are unchanged from the concrete rule. Other type conversion rules are defined in a similar way.

Currently, our abstract domains mainly consist of the definitions for arithmetic and logical operations together with type conversions. A notable exception is the presence of the `print`

operation (used internally to perform `echo` and `var_dump`). This allows us to treat output differently according to the chosen domain. For example, while performing an analysis we might not want to cause every `echo` statement to produce an actual visible output, as we might only be interested in calculating, for example, the sign of arithmetic variables. On the other hand, when performing taint analysis, we might want to raise a warning when a `print` statement is invoked with a tainted argument as input. Therefore, we consider `print` a domain-dependent operation, which we take the freedom to define differently in every domain, according to the specific analysis to be achieved.

While this process of redefining the necessary rules for every new domain might seem tedious, we point out that the current size (in terms of lines of code and number of rules) of our domain definition is very small compared to the overall size of $\mathbb{KPHP}$. For example, the sign domain currently consists of around 650 LOCs (370 for domain operations, 70 for abstraction function and 211 for lattice definition), while the full $\mathbb{KPHP}$ consists of over 6000 LOCs. We should also mention the fact that, for this simple domain, we defined operations such as the least upper bound in a purely tabular form (which wastes a lot of space) for the sake of simplicity and understanding. If we were to use a more concise form, the size of the domain would be even smaller. Furthermore, while the sign domain is a simplified example, the set of operations that would need to be redefined will be the same in all other domains, meaning that, even when considering a more advanced domain one only needs to provide a lattice, an abstraction function and appropriate definitions for arithmetic, logical, type conversion and input/output operations, while everything else is left unchanged.

7.2.5 Concrete domain

The process described above also applies to the concrete domain. In other words, the original arithmetic operations (as well as other rules, as described) are not to be discarded but instead should be added into an appropriate concrete domain definition, which should reside alongside other domains inside the domains folder. For the concrete domain, the abstraction function α is simply the identity. When the concrete domain is used, programs are simply executed according

to the standard behaviour defined in the original KPHP.

7.2.6 Example

With the above structure in place, KPHP can be compiled as usual with

```
kompile kphp.k
```

to obtain a simple static analyser that computes the signs of numeric variables and keeps strings concrete. At this stage it is not possible, in general, to correctly evaluate conditionals and loops in the presence of uncertain guards. Rules for conditional, loops, functions etc. will be introduced later. However, the reader should note that, unlike domain operations, control-flow rules are abstracted once and for all, and they will *not* have to be defined for every domain. Those updated rules will generalise the previous ones, in the sense that, in the presence of an abstract domain they will lead to a conservative approximation of the behaviours, while in the presence of a concrete/identity domain, they will model the actual behaviour (as previously defined in KPHP concrete rules).

However, provided that conditionals, loops and recursive functions are avoided, it is already possible to analyse PHP programs using the sign domain. For example, running the program `test.php` below

```
$x = 0;          // x : AIntZero
$y = 1;          // y : AIntPos
$z = 2;          // z : AIntPos
$t = $y + $x;    // t : AIntPos
$q = $y - $z;    // q : AIntTop
```

via the usual command

```
kphp test.php --config
```

produces a configuration which, when inspected, reveals that variables `$x`, `$y` and `$z` has been mapped to abstract values `Zero`, `AIntPos` and `AIntPos`, while variables `$t` and `$q` respectively contain

`AIntPos` and `AIntTop`. This is due to the fact that, according to this simple domain, the sum of a positive number with zero leads to a positive number, while the subtraction of a positive number from another positive number is unknown (last line, variable `$q`).

7.3 Lifting the semantics

In the previous section, we have refactored $\mathbb{KPHP}$ in order to make the domain of computation a parameter. In this simple setting, we are able to manipulate abstract values (by performing arithmetic operations and type conversions on them) and store them in memory via assignments. Also, we are able to pass and return those values from simple functions. However, problems arise when, for example, the guard of a conditional (or loop) becomes unknown, as in the following example

```
if (2 == 3)
        $x = 1;
else
        $x = -1;
```

where the comparison `2 == 3` is evaluated to `ABoolTop` in our simple sign domain, because it becomes `AIntPos == AIntPos` and therefore it is impossible to know the exact result, and which branch of the conditional to take. Therefore, we must adopt a conservative approximation, meaning intuitively that we shall execute both branches. In such cases, the existing semantics for the conditional operator cannot work, as it is defined as a simple case analysis on the value of the guard. If true, the first branch is taken, if false the second branch is taken instead. Similar issues arise in loops and recursive function calls, where we don't know how many times the function is going to call itself. In general, in the presence of uncertain control flow, we'd like to be able to run one or more computations in parallel and provide separate, precise analysis results. Unfortunately, this would lead to combinatorial explosion, so we need to conservatively *merge* the results of those pseudo-parallel branches. Consider again the above example, and suppose after the conditional we wish to inspect the content of variable `$x`. The boolean guard is unknown, therefore we have

to assume both branches could be taken, meaning that after the conditional the variable `$x` may contain `1` *or* `-1`. To compute this we need to evaluate both branches of the conditional in parallel, starting at the program state at the location immediately preceding the conditional, and finally merge the two obtained program states into one.

Our next step, described in this section, is then to "lift" or generalise our semantics so that it can correctly operate in such cases. Since our self-imposed goal is for our abstract semantics to be general enough to work correctly also on the concrete domain, we had to deal with the additional challenge of not being able to provide ad hoc abstract rules for the abstract case while maintaining the existing ones for concrete execution. Instead, we designed out abstract semantics as a proper generalisation of the concrete one, meaning that our abstract rules model both concrete and abstract execution, in a single framework. In other words, we designed our rules so that concrete execution (obtained by instantiating the abstract semantics with the concrete domain) is nothing but a corner case of the more general abstract semantics. This introduced of course the additional challenge of not being able to "forget too much information" and limited our freedom to design easier ad hoc solutions to particular analysis problems. On the other hand, we also see this as a sanity check: since our abstract rules are closely related to the concrete ones (which, indeed are particular cases of them), it becomes easier to debug and assess the soundness of our analysis framework by testing concrete examples.

In order to design the abstract semantics we use the conceptual framework of abstract interpretation, and base our core development (i.e. conditionals, while loops and memory filtering, detailed in the following) on well known results as presented in [94]. In the remainder of this section we detail the most important features of our abstract semantics - how they are obtained and how they work. We start from conditionals and loops and a discussion about how we deal with the problem of merging two program states (i.e. $\mathbb{KPHP}$ configurations) into one - this turned out to be especially challenging due to the presence of aliasing. We then move to the remaining major language features such as (recursive) function call, return, array and object access (including cases where the exact key to be accessed is unknown).

7.3.1 Conditionals

As a first step, we design abstract rules for the conditional statement, which in $\mathbb{KPHP}$ consist of a pair of standard rules conceptually equivalent to

```
rule if(true,TrueBranch,FalseBranch) => TrueBranch
rule if(false,TrueBranch,FalseBranch) => FalseBranch
```

A naive choice would consist in adding a third rule for dealing with uncertain guards, such as

```
rule if(ABoolTop,TrueBranch,FalseBranch) => merge(TrueBranch,FalseBranch)
```

where we can assume `merge` to be a procedure that runs both code fragments in pseudo-parallel and then combines the resulting states into one. However, as we mentioned, we aim at providing a single general rule to deal with both cases, so that the concrete semantics becomes an instance of a more general abstract semantics. We now detail the derivation of our abstract conditional rule in detail. Note that the following style of reasoning is then used in order to derive all other required abstract rules that constitute our abstract semantics.

Lifting the semantics and conditionals

We now perform the conceptual step of lifting our semantics, from a function $F : State \to State$ to a function $F : \mathcal{P}(State) \to \mathcal{P}(State)$. This is in fact the concept of concrete collecting semantics introduced in 4.5. Notice that no actual modification of the existing rules is required as this is only a conceptual step. In this setting, every language statement can be seen as a function taking a set of states as input and producing a set of states as output, where the output states are simply obtained by evaluating the statement on each of the input state, one by one

$$F(S) = \{s' \in State | \exists s \in S . s \Downarrow s'\}$$

Consider now the familiar conditional statement

```
if G
        TrueBranch
```

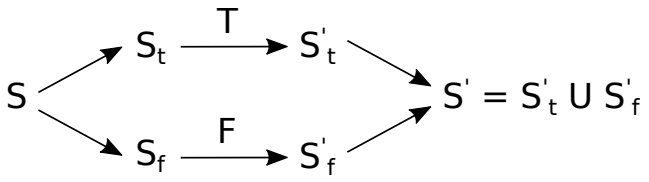


Figure 7.2: Evaluation of the conditional statement `if G then T; else F;` starting from the set of initial states S. The true branch T is evaluated for all initial states in which the guard evaluates to true. Conversely, the false branch F is evaluated in all states in which the conditional evaluates to false. The set of output states S' is the union of the result states of both branches.

```
else

        FalseBranch
```

In the concrete collecting semantics, the conditional is evaluated starting from *a set* of states, and shall therefore produce a set of states as result. Given an initial set of states S, this is naturally partitioned into two sets: the set of states in which the guard of the conditional evaluates to true, S_t, and the set of states in which it evaluates to false, S_f. According to the standard semantics of the conditional, when the guard is true, the true branch must be executed, and when it is false, the false branch must be executed instead. In our new setting, some of the input states would cause the true branch to be evaluated and some would cause the false branch to be evaluated. Therefore, the collecting semantics consists in evaluating in parallel the true branch starting from S_t and the false branch starting from S_f, and then performing a set-theoretic union of the outputs to obtain the final set of states. This is shown diagrammatically in Figure 7.2. Notice that this "lifted" behaviour is perfectly consistent with the standard behaviour. Indeed, a single concrete execution corresponds to the case where the set of initial states S contains a single state

$$S = \{s\}$$

In this case, the guard G must either evaluate to true or false in s, so that one of S_t or S_f must be the empty set. However, according to our formulation of the semantic function F, evaluating a statement starting from an empty set of states must in turn return the empty set (i.e. $F(\emptyset) = \emptyset$).

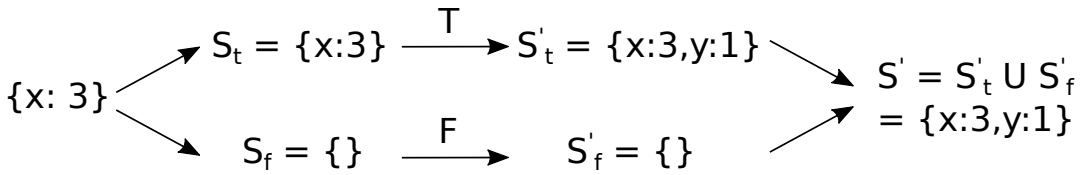


Figure 7.3: Standard conditional statement semantics as a corner case of the general abstract semantics

This means that in this particular case, only one branch will be evaluated, and the final state will be the output state that we would normally expect. Consider for example the "standard" execution of the following program

```
$x = 3;
if ($x == 3)
        $y = 1;
else
        $y = -1;
```

where the conditional statement is executed under one single state containing the variable `$x` whose value is 3. The evaluation of the conditional according to our generalised semantics is shown in Figure 7.3, demonstrating how the traditional semantics (as defined in $\mathbb{KPHP}$) is obtained as a corner case of the abstract semantics.

While it is not feasible in general to deal with sets of states (doing this would be equivalent to performing a sound and complete analysis, as discussed in 4.1), we aim to apply this idea when operating on abstract states. In this framework, in fact, we will not be dealing with sets of states explicitly, but instead we will consider single abstract states which in turn *represent* sets of concrete ones, as expressed by the concretisation function γ. Notice however that working with concrete sets of states containing a single element is indeed perfectly possible, and it is in fact what concrete execution means. In order to put the above concept into practice, we need to introduce the concept of *memory filter*.

Memory filter

Let's turn our attention once again at the concrete collecting semantics. Consider a set of states S and a boolean expression G. Then

$$filter(S, G) = \{s \in S.s \models G \Downarrow true\}$$

meaning that $filter(S, G)$ returns the subset of states of S in which the boolean expression G evaluates to true. If we had such an operation available, we could describe the semantics for conditional statements as

```
<mem> M </mem>
<k> if(G,TrueBranch,FalseBranch) =>
  setUnion(
    runInMemory(filter(M,G),TrueBranch),
    runInMemory(filter(M ~G)),FalseBranch) ...
<k>
```

where `setUnion` performs the union of two sets and `runInMemory` simply executes some code in the given state.

Notice that, while we introduced memory filters according to the classical treatment of [94], in general, when arbitrary expressions are allowed inside conditional guards, special care should be taken in order to avoid possible issues due to the way their side-effects are handled. One alternative formulation of memory filter, which avoids evaluating the guard twice is the following:

$$filter(S, G) = [\{s \in S.s \models G \Downarrow true\}, \{s \in S.s \models G \Downarrow false\}]$$

In the rest of this dissertation, however, we use the first formulation, for ease of exposition.

While $filter$ is not computable in general, we are able to provide a sound approximation of the desired behaviour when dealing with abstract states encoding sets of concrete ones. In order to accomplish this, we use the $isValid$ and $isUnsat$ predicates defined above. Intuitively, $isValid(B)$

expresses the fact that the abstract boolean B is *certainly* true, while $isUnsat(B)$ expresses the fact that it is *certainly* false. An abstract boolean B for which both $isValid$ and $isUnsat$ both evaluate to false expresses the fact that the actual concrete truth value of B is not known. For example, in our sign domain we have

```
rule isValid(ABoolTrue) => true
rule isValid(ABoolFalse) => false
rule isValid(ABoolTop) => false
```

and

```
rule isUnsat(ABoolTrue) => false
rule isUnsat(ABoolFalse) => true
rule isUnsat(ABoolTop) => false
```

while in the "concrete" domain (where abstract booleans coincide with the usual booleans) we simply have

```
rule isValid(true) => true
rule isValid(false) => false
```

and

```
rule isUnsat(true) => false
rule isUnsat(false) => true
```

We then define a simple overapproximation of the memory filter function as follows

$$filter(S, G) = \begin{cases} S & \text{if } isValid(G) \\ \emptyset & \text{if } isUnsat(G) \\ S & \text{if } \neg isValid(G) \wedge \neg isUnsat(G) \end{cases}$$

Notice how approximation (i.e. information loss) is introduced in the last case, where instead of trying to compute the appropriate representation of the subset of S for which the guard evaluates to true, we return S itself, meaning that the resulting set of states (or abstract state) is a superset of the actual set of states returned by the concrete, mathematical definition of the filter. Therefore,

our choice is sound. While more sophisticated options are available [94] we found that their inclusion in our development would dramatically increase the complexity of the codebase (especially in the presence of expressions with side effects) as well as the number of additional operations to be defined for every abstract domain. Moreover, while applicable in simple ad-hoc cases (such as for guards with consisting of simple arithmetic expressions with no side-effects) those techniques are no longer useful when guards can consist of *arbitrary* expressions with side effects, including function calls or even `eval`. Our choice represents therefore a compromise between the desire to improve precision and simplicity, and in particular enables us to maintain enough precision to be able to deal with concrete executions while maintaining the development simple by avoiding the need for backward semantic rules [94]. Most related work (e.g. [106] and [107]) uses a similar approach, although formulated in terms of `assume` statements, while in our development we follow the classical presentation of [94]. When `assume` statements are used, both branches of the conditional are evaluated, but an `assume(G)` statement (where `G` is the guard of the conditional) is inserted at the beginning of each branch. The evaluation of `assume(G)` consists in deciding wether `G` is satisfiable or unsatisfiable, similarly to our approach. If `G` is unsatisfiable, `assume(G)` cause the branch to be skipped, otherwise the branch is evaluated. Moreover, in case `G` is satisfiable, `assume(G)` should also *refine* the current state by incorporating facts implied by `G`, similarly to the classical formulation in term of memory filter. However, none of the literature we inspected did in fact explain how this is achieved in practice (in the general case, this would amount to perform the full backward-directed analysis mentioned above). We believe that, in most cases, a compromise is made; either the state is generally not refined (unless `G` is unsatisfiable, in that case the branch is not executed at all), or some restricted form of refinement is performed, for example by using type refinement on union types as in PHANTM [106] (discussed in Section 4.7). Alternatively, some tools always consider both branches of the conditional and don't perform any type of memory filtering at all. In RIPS [104], for example, both branches are evaluated; the guard of the conditional however, is inserted at the beginning of both branches, so that any side effect is also taken into account during the simulation.

The abstract conditional

We are now able to define the semantic rule for abstract conditional.

```
rule [if]:
  <k> 'If(C,,B1,,B2) =>
    mergeConfigs(
      runAndGetConfig(runInMem(filter(C, some(E)), B1)),
      runAndGetConfig(runInMem(filter('Not(C), some(E)), B2))) ... </k>
 <mem> E </mem>
```

where `mergeConfigs` provides the abstract version of set union (and will be discussed in great detail in a later section).

7.3.2 While loop

We now discuss the derivation of our abstract semantics rule for the `while` loop. As will become evident, we re-use the concept developed for the conditional. In particular the abstract semantics for the while loop also relies on the use of a *memory filter* in order to allow the concrete execution case to simply become a particular case of the more general semantics.

The collecting semantics of the while loop

Let us consider the construct

```
while G do B;
```

where `G` is a boolean expression and `B` is the loop body, and let us assume, as done in the previous derivation of the conditional statement, that the value of the guard `G` is not known a priori, or that its value is an abstract boolean encoding uncertainty (e.g. `ABoolTop`). Again, remember we are now in a setting where we assume the semantics to be a function from sets of states to sets of states. In order to provide a conservative estimate of the set of states *after* the conditional has been executed, we have to assume that the loop body might not be executed *or* that it might be executed once, or twice etc.; since the truth value of the guard is unknown, the loop body might

actually be executed *any* number of times. In other words, if the initial set of states is S, and we encode the loop body `B` as a function

$$B : \mathcal{P}(States) \to \mathcal{P}(States)$$

then, ignoring the presence of side effects, the final set of states S' can be expressed as

$$S' = S \cup B(S) \cup B(B(S)) \dots$$

i.e. the final set of states S' consists of all the possible states that could be obtained by not running the loop body, or by running it $1, 2, \dots$ times. If we consider instead the presence of side effects, the above expression becomes

$$S' = G(S) \cup G(B(G(S))) \cup G(B(G(B(G(S))))) \dots$$

in order to take into account the evaluation of the guard G at every iteration. S' can in fact be expressed as the *least fixpoint* of a function

$$g^0(S) = G(S)$$
$$g^{n+1}(S) = G(B(g^n(S))) \cup S$$

meaning that

$$S' = \bigcup_{i=0\dots\infty} g^i(S) = lfp\ g$$

The collecting semantics, revisited

What we have just outlined is the standard definition of the abstract semantics of the while loop. However, providing an abstract semantics that just computes the fix point of the loop body (without even considering the guard) is not enough for us, because we need that extra bit of

precision necessary to allow us to also handle concrete execution. In other words, we want to be able to perform the fix point computation when the guard is uncertain, while still being able to compute the exact result where the truth value of the guard is known. Furthermore, we want to do this in the most general way, without artificially distinguishing two different cases, such as a standard rule for concrete domain and fix point rule for abstract ones.

To derive our rules, we start by reasoning in the context of a concrete collecting semantics, as done for the conditional. We will then derive a rule which works for any abstract domain. Consider an initial set of states S, and again our statement

```
while G do B;
```

Similarly to what we have previously seen, the guard `G` will evaluate to true for a certain subset S_t of states of S and to false for a subset S_f. The loop body `B` is then evaluated from the initial states S_t. The set S_f, on the other hand, is left unmodified because the loop body is not evaluated (since `G` evaluates to false in all of those states). We can then merge the two sets S_t and S_f (via a set-theoretic union or its abstract version given by the merge mechanism) to obtain the set S' containing all possible states obtained by running the loop body zero *or* a single time. We call this process a *loop step*. Now, the process could easily be repeated starting from S', which will in turn be partitioned into two subsets, one for which the guard evaluates to true and one for which the guard evaluates to false. The above steps are then repeated to obtain a second set S'' containing all the possible states obtained by running the loop body zero, one or two times. The computation ends when a fixpoint is reached, which could easily be detected by the fact that, at some point,

$$S^i = S^{i+1}$$

This is shown diagrammatically in figure 7.4. As for the conditional case, the partitioning of sets of states according to wether a conditional express evaluates to true or false is done via our memory filter function.

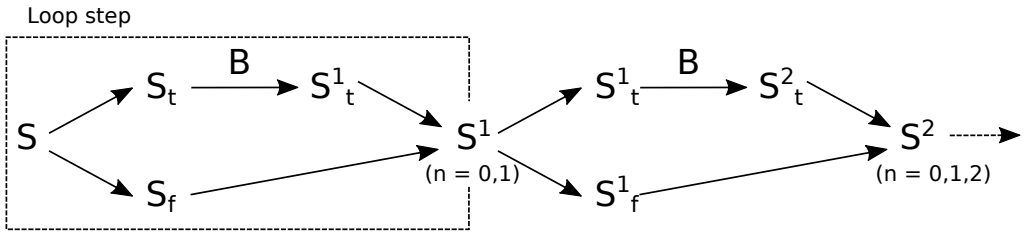


Figure 7.4: Abstract evaluation of a while loop. At every step, the initial set of states is partitioned according to wether the guard evaluates to true or false (as also done in the conditional). The loop body is then evaluated once, starting from the set of states that validates the guard. The final set of states in then merged with the set of states for which the guard is false. The process is repeated until a fixed point is found.

Abstract semantics of the while loop

We are now ready to show our $\mathbb{K}$ implementation of the abstract semantics of the while loop. The top-level rule is simply defined as

```
rule
  <k> 'While(B:K,,S:K) =>
    LFP(none, #whileStep(B,S)) ~> moveOnFromLoop(B) ... </k>
    [transition]
```

expressing the fact that the semantics of a while loop with guard `B` and loop body `S` is obtained by computing the fixpoint of the iterated execution of single while-steps, as discussed above. The `LFP` internal function takes care of computing the least fixed point, where the first parameter, initially set to `none`, represent the previous state to be compared with the output one. Notice however that, in order to compute the least fix point, memories are compared modulo *isomorphism*, as we'll discuss again in the remainder of this Chapter. Essentially, we consider two memories to be isomorphic when they are the same modulo the *naming* of memory locations. In other words, when comparing memories we ignore the implementation-dependent detail of memory allocation, focusing only on their content and structure.

The `moveOnFromLoop` auxiliary operation takes care of a special case and will be discussed in the following. The `whileStep` operation computes the output state after a single while-step as

discussed above, and is defined just like the conditional discussed above in this Section:

```
//@ compute a "single step" of while loop
syntax K ::= "#whileStep" "(" K "," K ")"

rule
  <k> #whileStep(C,S) =>
    mergeConfigs(
      runAndGetConfig(runInMem(filter(C, some(E)), S)),
      runAndGetConfig(runInMem(filter('Not(C), some(E)), .K))) ... </k>
  <mem> E </mem>
  [transition]
```

This relies on the `mergeConfig` operation, which will be detailed in 7.4. For now, the reader just notice that `mergeConfig` essentially performs the role of the set-theoretic union we used during the derivation of the rules, where instead of set union the least-upper-bound operator is used so that the resulting merged state is a single abstract state encoding a sound approximation of a concrete set of states.

Once again, we stress the fact that the abstract semantics of the while loop as defined here strictly generalises the standard semantics. As an illustrative example, let us consider the simple program

```
$x = 2;
while ($x > 1)
        $x = $x - 1;
```

Figure 7.5 shows the concrete/standard execution of the loop under the new semantics. Notice that, in the concrete case, merging of states always happens between *trivial* sets (i.e. the empty set). This means that we can effectively mimic the standard execution while maintaining a unified theoretical understanding of the semantics.

An interesting anomaly (and how we deal with it)

Consider a very simple program such as

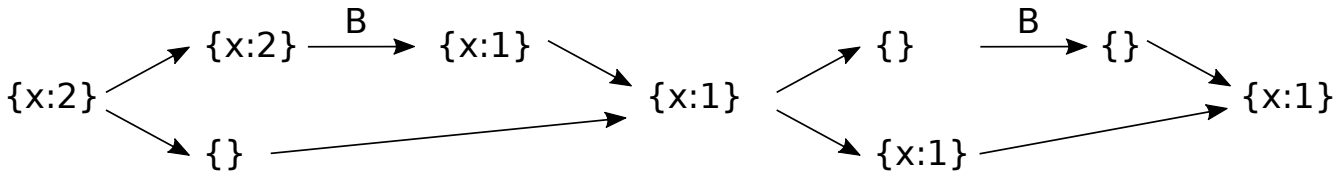


Figure 7.5: Standard execution of a while loop under the new fix-point based semantics. The diagram shows that the loop body is executed only once as expected, and a fixpoint is reached since $S^1 = S^2$.

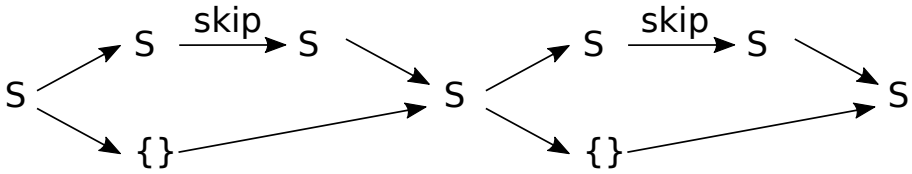


Figure 7.6: Executing a trivially looping program which does not update the program state causes the loop to terminate as a result of reaching a fixed point.

```
while (true)
        skip;
```

This program should obviously loop in the concrete domain, since the guard is explicitly `true`. However, evaluating such a program in our updated semantics causes its immediate termination. This makes sense, since the loop body is not changing the program state and therefore the computation will immediately reach a fixed point. This behaviour is shown if figure 7.6. While this might initially look like a mistake, we believe it's not. In fact, we are able to detect this situation in an easy way, by simply performing a check on the loop guard *after* the loop has terminated. If the predicate $isValid(G)$ returns true, this means the guard was still "definitely true" after the loop, but since the loop actually terminated this has to be caused by the fact that the loop body didn't change the state in any way. We perform this check via the `moveOnFromLoop` auxiliary operator that, we recall, appears immediately after the fixpoint computation in the semantics of the while loop.

```
rule
  <k> 'While(B:K,,S:K) =>
```

```
    LFP(none, #whileStep(B,S)) ~> moveOnFromLoop(B) ... </k>
    [transition]
```

The `moveOnFromLoop` operator simply computes the value of the $isValid$ predicate for the loop guard. If its value is true, then we add a special `LOOP` flag to the `<status>` cell, otherwise we insert the standard `NEXT` flag and continue execution as normal:

```
syntax  K ::= "moveOnFromLoop" "(" K ")" [strict]

rule
  <k> moveOnFromLoop(V:KResult) => . ... </k>
  <status> _ => NEXT </status>
  when notBool (valid(V))

rule
  <k> moveOnFromLoop(V:KResult) => . ... </k>
  <status> _ => LOOP </status>
  when valid(V)
```

A `LOOP` status indicates the fact that an infinite loop has been encountered during the computation, and causes the subsequent program instructions to be ignored. Conversely, evaluating in the concrete domain trivially looping programs that do modify the state such as

```
$x = 1;
while (true)
        $x = $x + 1;
```

will indeed cause the program to loop, since at every loop step the resulting state will be different from the previous and the fixed point will never be reached.

About termination

This is probably the best place to clarify that, as indicated by what we have just discussed, our static analyser does not necessarily enforce termination. Indeed, it is up to the chosen domain to guarantee termination by making sure that a fixpoint is reached in a finite number of steps, either

by being a finite domain (as our sign domain) or by implementing some form of widening [94]. On the other hand, our concrete domain, for example, does not guarantee termination (as discussed above). Moreover, since our sign domain abstracts integers to their sign but keeps strings concrete, this may also cause non-termination, for example in cases where a string is continually concatenated with another in a loop. This is in line with our philosophy of being able to re-obtain the concrete behaviour as a particular instance of our generalised abstract semantics. Moreover, we also aim at giving each domain as much freedom as possible in terms of design choices (including what to keep concrete and what to abstract), so that analyses could be easily tailored to specific applications.

7.3.3 Functions and recursion

Also the semantics of functions needs to be adapted in order to deal with potentially unknown control flow. The return mechanism has to be generalised in order to take into account the possibility that, when exploring different paths, more than one return value is chosen, while the call mechanism needs to be adapted in order to deal with recursive functions. In fact, when the condition determining wether a function should recursively call itself is not known, we end up in a situation similar to that of the while loop: we don't know how many times the function will call itself and therefore me must over approximate the behaviour by assuming it may call itself $0, 1, \ldots n$ times. In the following we discuss how we deal with both situations.

Abstracting the semantics of return

Suppose a function contains conditionals or loops in its body, and one or more branches contain a return statement. In our new setting, we cannot simply return a value and resume the previous computation (as in standard semantics) as soon as a return statement is encountered, because we have to explore the other branches first. Therefore we need to change the semantics of return. Consider the example code

```
function foo() {
        if (?)
```

```
                return 1;
        else
                return -1;
}
$y = foo();
```

where the guard is unknown, expressed by using a question mark. The evaluation of the conditional amount to executing both branches and then merging the results, as seen above, however the first encountered `return` statement would indeed cause the function to return, so that the remaining computation (including the remaining part of its body) is lost and replaced with the one saved in the stack (see 5.2.4 for the semantics of functions in $\mathbb{KPHP}$). In practice, this will cause either `1` or `-1` to be returned (depending on which order the two pseudo-parallel computations are performed), which is unsound since it is not known which one of the two branches will be taken at runtime. Instead, the function above should return, as a conservative approximation, a set consisting of both values, or its abstract counterpart consisting on a single abstract value.

In order to implement this behaviour, we simplify the semantics of `return` itself so that all it does is simply to accumulate the returned value (using the least upper bound defined in the current domain) into a special `<returns>` cell that we appositely include in the configuration. Also, `return` updates the configuration status flag to `RETURN`, causing the subsequent instructions in the current branch to be ignored (essentially by evaluating them as "no-operations") while still maintaining the flow of execution so that other branches of the same function can be explored as necessary.

```
rule [return]:
  <k> 'Return(V:PlainLanguageValue) => . ... </k>
  <status> _ => RET </status>
  <returns> R => lub(R,V) </returns>
  [transition]
```

Once the whole function body with all its possible branches has been explored, the auxiliary operation `POP` takes care of returning the accumulated value and performing the usual operations necessary to return from a function (such as restoring the current scope, popping the stack frame

etc., see 5.2.4). We don't show the `POP` rule here as it is essentially identical to the `return` rule shown in 5.2.4, except for the fact that instead of returning a value given as argument, the value to be returned is retrieved from the `<returns>` cell.

A digression: status codes and sequential composition

We have just mentioned the fact that the `return` rule for abstract semantics updates the status the configuration status flag to `RETURN` and that this in turn causes all subsequent statements of the current branch to be ignored. This mechanism is implemented by equipping the configuration with an additional control cell `<status>` and by updating the sequential composition rules to that statements are either executed or ignored (i.e. treated as a skip or no-op operation) depending on the current status flag. In particular, when the status is `NEXT`, instructions are normally executed, while when it is `RETURN` are treated as no-operations. Therefore, our key rules also need to take care of setting and resetting the status flags appropriately.

Function call

The function call semantics needs to be updated in order to deal with cases where the number of (mutually) recursive calls is not known in advance. In such cases, we have to conservatively assume the function might recurse $0, 1, \ldots n, \ldots$ times. For the while loop, we dealt with this possibility by modelling its semantics as a fixed point computation, so that the output is a conservative approximation of the set of states which could be obtained regardless of the actual number of times the loop body is evaluated.

Our goal is therefore to model an analog behaviour for recursion, but unfortunately, for functions, we must deal with a few additional challenges. First at every call the stack is growing (since a new stack frame is pushed into it) meaning that, strictly speaking, a fixed point is never reached. Second, a while statement is clearly structured (i.e. there is a guard and a loop body) while recursive and mutually recursive function call can happen in a more unstructured way and in different settings. We develop our own "weaker" concept which resembles that of a fixed point,

and is suitable to approximate the behaviour of recursive functions in the presence of abstract values.

We build our abstract semantics of function call inspired by a simple idea: since functions are deterministic, *same input equals same output*. That is, each time a function call is attempted, we first check wether the same function has already been called (and hasn't returned yet) *from the same program point* and starting from an environment (including parameters, caller, global and superglobal scopes etc.) isomorphic to the current one. If this is the case, we avoid performing the actually call, because for the "same input equals same output" principle, this would produce the same output as the previous call. Instead, when this happens we directly return the value accumulated in the `<returns>` cell with the effect of causing the current function to return, without performing a new call. We see this as a fixpoint-like concept, and we say that the recursive function call has reached a fix point and therefore it can be considered terminated.

We implement this behaviour by adapting the existing internal operation `runFunction` (discussed in Chapter 5) so that the function call is performed conditionally, in line with the strategy outlined above. In particular, before performing the actual call and pushing a new frame into the stack, we create the stack frame we wish to insert (but we don't push it into the stack yet), augmented with a label identifying the call site and the function's name and we compare this new frame against all the stack frames already in the stack, in order to see if it is *similar* to any existing one. Two stack frames are similar iff

- their call site labels and function names are the same
- the associated heaps are *isomorphic*.

Again, recall that two heaps are isomorphic when they are the same heap *modulo* the naming of memory locations, which is an implementation-dependent detail. If the stack frame we are considering for insertion is not similar to any stack frame already in the stack, then we call the function as usual, by performing the standard operations (pushing the frame into the stack, initialising a fresh scope with the parameters and eventually running the function body). Otherwise, the function

is not called and instead the accumulated value present in the `<returns>` cell is returned. Notice that, during initialisation, the `<returns>` cell is initialised with the lattice infimum (usually the value `Bot`); if a recursive function never returns, then this default value it returned, consistently with the intuition that the function is not returning any value at all.

This behaviour is achieved by splitting the existing `runFunction` rule into two (each modelling one of the two alternative behaviours), and by defining an appropriate predicate that performs the required checks as a side condition. The following rule defines the case in which the function is executed.

```
rule [run-function]:
  <k> runFunction(FName:String, f(Parameters:K, Body:K, RetType:K, LStatic:Loc), Args:K,
      Class:OptionId,Obj:OptionLoc,Static:Bool,Lab,Sf:StackFrame) ~> K:K =>
  *Cond(
    (notBool Static) andBool ((Obj ==K none) andBool (CurrentObj ==K .K)),
    NOTICE("Non-static method should not be called statically\n"),.) ~>
    processFunArgs(Parameters, Args) ~>
    pushStackFrame(Sf) ~>
    ArrayCreateEmpty(L1) ~>
    setCrntScope(L1) ~>
    incRefCount(L1) ~>
    setCrntClass(Class) ~>
    *Cond(Obj =/=K none, allocCrntObj(Obj), .K) ~>
    CopyFunctionArgs ~>
    'ListWrap(Body,, 'Return(NULL)) ~> POP(K) </k>
  <functionArgumentsDeclaration> D:K => . </functionArgumentsDeclaration>
  <currentScope> L:Loc </currentScope>
  <heap> H:Map </heap>
  <thematrix> H' </thematrix>
  <class> CurrentClass:K </class>
  <functionStack> S:List </functionStack>
  <object> CurrentObj:K </object>
  when (fresh(L1:Loc)) andBool (notBool((H',H) |- Sf isIn S))
```

```
  [internal]
```

Note that the only differences with the original rule (see 5.2.4) is the side condition `notBool((H',H) |- Sf isIn S)` which states that the stack frame `Sf` is not present yet in the stack `S` (modulo naming of memory locations) and the fact that, in the K cell, after the function body we insert another item `POP`. As discussed, `POP` performs the operations necessary to return from the function, and it is equivalent to `return` in the original semantics. However, by scheduling `POP` to be executed at the very end of the function body and by modifying the semantics of `return` as explained below, we ensure that all possible return values are accumulated *before* performing the actual return (causing the stack frame to be discarded and the previous computation to be resumed). Finally, for the converse case, the rule is simply defined as

```
rule [run-function]:
  <k> (runFunction(FName:String, f(Parameters:K, Body:K, RetType:K, LStatic:Loc), Args:K,
    Class:OptionId,Obj:OptionLoc,Static:Bool,Lab,Sf) ~> K:K) => R ~> K </k>
  <returns> R </returns>
  <heap> H:Map </heap>
  <thematrix> H' </thematrix>
  <functionStack> S:List </functionStack>
  when (H',H) |- Sf isIn S
  [internal]
```

We now clarify these ideas with a concrete example. Consider the following fragment of code, a standard factorial function where we added two comments `(1)` and `(2)` to mark the different call sites - one at top-level and one inside the function itself where the recursive call is performed (in our $\mathbb{K}$ development we add such labels to every function call AST node).

```
function factorial($n) {
        if ($n == 0)
                return 1;
        else
                return factorial($n - 1) // call site (2)
}
```

```
factorial (6);  // call site (1)
```

Figure 7.5.2 shows the key steps in its evaluation in our sign domain, and how at the third execution a fixpoint is reached, so that recursion is stopped and the accumulated value `AIntTop` is returned. In more detail, execution proceeds as follows. The first time, the function is called from the top level (location `(1)`). Since it is the first call, the normal operations of pushing a stack frame into the stack and eventually executing the function body are performed. Moreover, the local variable `n` is assigned a value `Pos` ($\alpha(6)$), which becomes part of the local environment (expressed as `n:Pos` in the figure, and depicted as a rounded rectangle next to the stack frame).

Then a first recursive call is attempted. Also this time the call is performed normally, because it is the first time the function is being called from the call site `(2)` and therefore there is no other stack frame containing that same label. Notice that, in the caller, we have `n:Pos`, but, since in the sign domain it is $\texttt{Pos} - \texttt{Pos} = \texttt{Top}$, the value `Top` is passed in the call (this is again shown next to the newly inserted frame in the diagram).

At this point, another recursive call is performed, from the same call site `(2)`. However, notice that while there exist a stack frame currently in the stack referring to the same call site `(2)`, those two differs in their associated memory, as clearly visible in the diagram. In fact, the currently candidate stack frame is linked to an environment where `n` is `Top` where in the one associated to the existing stack frame, `n` is actually positive. Since, for this reasons, no existing stack frame is *similar* to the candidate, we push it onto the stack and perform function call as usual.

At the next call, however, we detect that the candidate stack frame is indeed similar to its predecessor (shown in red in the figure). For this reason, the second case of `runFunction` is applied, causing the recursive call to be ignored and instead the already accumulated value `Top` to be returned.

To the best of our knowledge, we are the first to fully support recursion in a PHP static analyser. As mentioned in 4.7, most PHP static analysers ignore recursive calls (e.g. [77]) or deal with them in a restricted way (e.g [104]), where the actual nature of those restrictions is not detailed. Other authors simply don't mention wether their tools support recursion or not

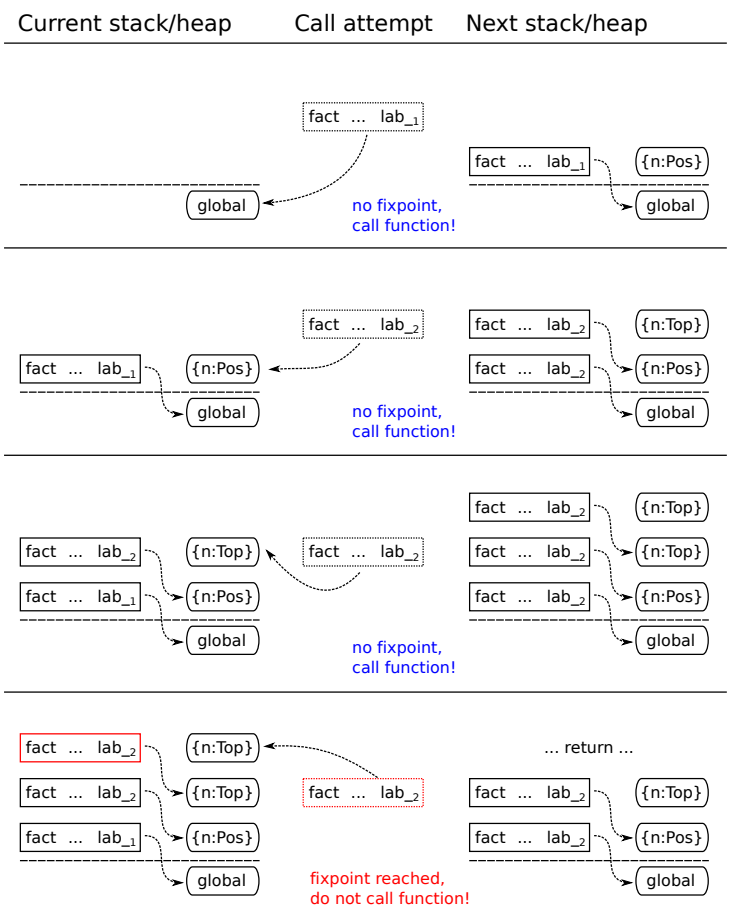


Figure 7.7: Example of recursive function call. After the second recursive call, a fix point is detected by inspecting the stack and noticing that a stack frame *similar* to the one being inserted is already present.

(e.g. [124, 106, 125, 126]), hence we were not able to perform an accurate comparison. However we believe that, since recursion is an important and challenging topic, if those tools provided a novel solution to the problem, this would certainly have been mentioned in the literature.

Notably, Pixy is the only tool for which the way of handling recursion has been (briefly) discussed in the associated literature. In [76], Jovanovic *et al.* mention that "*every instance of a called function contains its own copies of its local variables (variable incarnations). In most cases, it is not possible to decide statically how deep recursive call chains can become since the depth may depend on dynamic aspects, such as values originating from databases, or user input. Hence, static analysis would be faced with an infinite number of variable incarnations*". Although they don't detail they way in which they deal with the issue, they briefly mention that, in Pixy, "*inside*

functions, the analysis only tracks information about global variables and its own local variable incarnations.". Their approach is therefore similar to ours in spirit, even though we were not able to assess it in detail due to a lack of an in-depth description in the literature.

Variable Functions

PHP supports *variable functions*, i.e. function calls in which the name of the function to be called is obtained by evaluating a variable:

```
$fun = "foo";
$x = $fun();
```

In order to deal with this, we further generalise the semantics of function call (discussed above) in order to deal with *sets* of functions instead as with a single functions. Consider for example the code

```
$fun = ? // $fun is unknown
$x = $fun();
```

where variable `$foo`, which contains the name of the function to be called, is statically unknown (for example, it may have been evaluated to `AStringTop` in the signs domain). In this case, our static analyser must assume that every defined function *may* be called. The set of possibly called functions is determined by inspecting the `<functions>` cell and extracting the set of function names which are smaller then (according to the lattice ordering of the current domain) the target (`AIntTop` in this case). Once a set of functions is determined, all of them are called in pseudo-parallel (via the semantics discussed above), and the results are summarised using the existing merging mechanism. When instead the function name is exactly known (as in the first example above) the set of possibly called functions becomes a singleton, and only the desired function is called. This confirms that the generalised abstract semantics correctly reproduces the concrete semantics. Moreover, notice that the precision of the approach depends on the chosen domain. In the example above (evaluated in our rather imprecise signs domain) variable `$fun` evaluated to a "completely unknown" value and therefore all possible functions had to be called. If we were in a more precise domain, such

as one that tracks sets of strings, then we would be able to consider a smaller set of candidate functions. For example, in the following program

```
if (?)
        $fun = "foo";
else
        $fun = "bar";
$x = $fun();
```

only two possible calls (`foo()` and `bar()`) would be considered assuming that variable `$fun` evaluated to the set $\{foo, bar\}$ after the conditional. In our signs domain instead, the conditional above causes `$fun` to directly evaluate to `AStringTop`, causing all functions to be called as seen above.

Break and Continue

Similarly to the semantics of `return` (discussed above), also the semantics of `break` and `continue` needs to be updated for analogous reasons. Consider for example a situation in which one branch of a conditional contains a break but the other doesn't:

```
while (?) {
        if (?)
                break;
        else
                $x = -1;
}
```

If we were to keep the original semantics for `break` (see 5.2.4) the statement would cause the execution of the loop to be aborted immediately, without allowing the other branch to be considered.

Therefore, as a first step, we update the semantics of `break` so that instead of aborting the loop, it simply adds a `BREAK(n)` status code to the `<status>` cell, where the argument `N` is the argument provided to the `break` statement (recall that if no argument is provided, as in `break`, then this is equivalent to `break(1)`):

```
rule [break]:
  <k> 'Break(N:Int) => . ... </k>
```

```
  <status> NEXT => BREAK(N) </status>
  [step]
```

As the status code `RET`, activated when a function returns, also `BREAK(N)` is defined as a *skipping code*, meaning that, when present in the configuration, all user program statements will be skipped, ensuring that the continuation is not discarded but also that the internal bookkeeping operations are still performed.

The second step consists in updating our loop constructs so that, when the loop terminates, they update the configuration's status code accordingly. In particular, the `BREAK(N)` is updated as follow:

- If `N` $= 1$ then the new status code becomes `NEXT`
- If `N` > 1 then the new status code becomes `BREAK(N - 1)`

This is achieved by the `decreaseBreak` internal operation

```
rule [decrease-break-N]:
  <k> decreaseBreak => . ... </k>
  <status> BREAK(N:Int) => BREAK(N -Int 1) </status>
  when N >Int 1

rule [decrease-break-1]:
  <k> decreaseBreak => . ... </k>
  <status> BREAK(1) => NEXT </status>
```

invoked at the end of every loop. For example, in the case of the `while` loop, the semantics is updated as follow:

```
rule
  <k> 'While(B:K,,S:K) =>
    LFP(none, #whileStep(B,S)) ~> moveOnFromLoop(B) ~> decreaseBreak ... </k>
    [transition]
```

Finally, we shall consider how we deal with situations in which different branches are considered, each with a different level of `break` or even a combination of a `BREAK(N)` outcome and a different

one, such as `RET`, `NEXT` etc. We consider two cases. In the first, the two branches being considered have both executed a `break` statement, with same argument N:

```
while (?) {
        if (?) {break(1);}
        else {break(1);}
}
```

resulting in a call to `mergeConfigs` where both input configurations have the status code `BREAK(N)`. We deal with this case easily, by applying the same principles discussed before, i.e. the two configurations are merged into one before evaluating the next instruction. Consider instead a case where only one branch executes a `break`, or where both branches executes a `break` but with different arguments, such as:

```
while (?) {
        while (?) {
                if (?) {break(1);}
                else {break(2);}
        }
}
```

In this cases, it is not immediately obvious what strategy to use in order to merge the two different branches in a sound manner. To maintain soundness, we bypass the problem by *postponing* the merge operation: instead of combining the outcomes of the two branches into a single configuration before executing the next instruction, we essentially split the remaining computation into two pseudo-parallel threads. That is, the remainder of the computation is evaluated twice, in each of the two configurations, and therefore produces two different outcomes. Only at the end, those two outcomes are merged back into one, resulting in a sound approximation of the two alternative behaviours. At the time of writing, we are in the process of designing and implementing a more efficient approach to the problem.

As for recursive functions, our way of handling `break` and `continue` in a PHP static analyser is novel, as no related work addressed the issue so far (to the best of our knowledge). For example,

Xie and Aiken [77] claims that, in their analysis, *"a statement can be an assignment, function call, return, exit, or include"*, which implicitly mean that `break` is not covered (and indeed, it is not mentioned in the paper). In [126] instead, even if `break` and `continue` are not discussed explicitly, the authors mention that *"every case inside a* `switch` *must end with a* `break`*, so no fall-through is allowed"*. This leads us to think that their tool provides some form of restricted support for `break` needed in the modelling of `switch`. We were not able to find further information about the handling of `break` and `continue` in existing PHP static analysers.

7.4 Merging configurations

In the previous sections, we have discussed the abstract semantics of `if`, `while`, `break` and function call and return. In all those cases, we were relying on a certain numbers of internal operations which we haven't detailed yet. While some of them are pretty much self-explanatory and easy to define, others - and in particular `mergeConfigs` - are in fact quite the opposite and deserve a proper discussion. The `mergeConfigs` internal operation is in fact at the heart of our system, as it takes two $\mathbb{KPHP}$ (abstract) configurations and merges them back into a single one. The result is an (abstract) $\mathbb{KPHP}$ configuration that encodes a sound approximation of the set consisting of the two initial configurations. In order to accomplish its task, `mergeConfigs` relies on operations defined in the currently loaded domain (such as least-upper-bound).

The core idea behind the merging mechanism is simple. Consider the program

```
if (?)  $x = 1;
else $x = -1;
```

According to the semantics of the conditional statement (derived in 7.3.1), its evaluation amounts to executing the two branches in pseudo-parallel, then passing the two resulting configurations to the `mergeConfigs` procedure. Among other things, mergeConfig would recursively inspect the heap to determine which variables are defined in each branch and what values they contain, and finally combine the gathered information in order to build the resulting heap. In the example, the two

configurations to be merged are identical except for the fact that in one, `$x` contains the value `1`, while in the other one it contains `-1`. The resulting configuration will therefore contain the same variable `$x`, but the value assigned to it would be the least upper bound of the two possible values (e.g. `AIntTop` in the simple sign domain).

7.4.1 Challenges

The previous example was a simplified one, where all the details of the intricate PHP semantics were hidden.

Dealing with implementation-dependent allocation strategy

A first source of problems lies in the fact that we have to deal with the implementation-dependent naming of memory locations. As seen, variables in ($\mathbb{K}$) PHP are not directly mapped to values, but instead are mapped to locations which in turn are mapped to values. In $\mathbb{KPHP}$, when assigning a variable for the first time, we allocate it into a fresh memory location (as returned by the fresh builtin predicate of $\mathbb{K}$, but other strategies are equally possible) before writing the desired value to that location.

Consider again the previous example where `$x` was initialised in both branches of the conditional. The evaluation of the conditional (in the abstract semantics) would cause both branches to execute in pseudo parallel, starting from the program state existing just before the conditional itself, when variable `$x` was not defined. This means that, in both branches, `$x` will not be present at the time the assignment is reached, and this will cause the standard assignment and initialisation rule to be applied, causing `$x` to be created "from scratch" in both branches, independently from each other. Since in our allocation strategy we use a global built-in integer counter, variable `$x` will be associated to a different memory location in the two different branches. This fact has to be taken into account both when merging configurations and when comparing them.

Array-centric memory model and multiple entry points

As discussed in 5.2.1, the memory model of (𝕂) PHP is strongly array-centric, meaning that everything is essentially built on arrays: not only objects, but also scopes (both global, super global and local). In particular, what in traditional terminology could be called a *global environment*, in 𝕂ℙℍℙ is simply represented as the array of global variables, meaning that "merging two heaps" after executing a simple program without functions means essentially merging their two resulting global arrays. Moreover, suppose a program defines an array such as `$x = array("foo"=> 2, "bar"= 2;)` and therefore, at a merging point, a user array would have to be merged; This is in fact no different from merging "special" arrays such as the ones containing the super globals, globals or local variables. Therefore, as the previous discussion suggests, our `mergeConfig` operation strongly relies on the `mergeArray` one, which we use to merge both user and internal arrays.

Having clarified this, an additional layer of complexity if introduced by the presence of multiple *entry points* to the heap. Consider a generic 𝕂ℙℍℙ configuration and the execution of a generic program. Not only there are global and super global variables to consider, but also local function or method variables, static variables (globals or per-function), scopes of functions which are currently waiting from other calls to return and so on. When merging two configurations, all of this must be taken into account if we aim at achieving soundness. In practice, this amounts to calling our `mergeArray` procedure multiple times, one for each of those entry points. However, one more difficulty is introduced by the fact that each of this environments (each identified by an array which we consider an an entry point to the heap) may not be, in general, disjoint from others. For examples, variables may be passed *by reference* to a function, causing an aliasing between one or more variables in the caller and callee scopes. For this reason, we need to take extra care when invoking multiple instances of mergeArray in bulk, in order to make sure that every subsequent call does not loose any information about the information manipulated in the previous calls and that the possible aliasing relationships are preserved.

User-defined aliasing

Related to the problems discussed above, is the fact that aliasing may not only be present as a result of, for example, passing variables by reference, but it is also directly available to users as a language construct. This introduces the additional challenges of dealing with variables which *may* be possibly aliased (e.g. when aliasing is present in one branch but not in the other) in a sound way.

While the literature provides us with standard general-purpose techniques for *heap*, *points-to* or *alias* analysis (such as [127] and [128]), we found those not to fit our needs (for example, most heap analyses are flow insensitive) and in particular not suitable to fulfil our self-imposed requirement that that our abstract semantics should strictly generalise the concrete one. In other words, if we were to simply implement one of those standard analysis, we would not be able to re-obtain concrete execution when instantiating the analyser with a concrete domain. Therefore, we developed our own analysis strategy for dealing with the presence of aliasing, providing *full support* for aliasing and references, whereas most existing PHP analysers don't. One such case is Pixy [76], which performs alias analysis, but only on "non-array variables", meaning that, for example, the aliasing in `$x =& $y` would be tracked while the aliasing in `$x =& $y["foo"]` would be ignored. Other tools, such as [77] and [106] do not provide support for aliasing at all, as it is considered too challenging or expensive.

The presence of "ghost references"

As seen, our merging mechanism needs somehow to abstract from the internal, implementation-dependent representation (or naming) of memory locations. In particular (as we'll see later), it may happen that, in the heap resulting from a merge, variables will be re-allocated into new memory locations (while, obviously, still preserving their relationships and the overall shape of the heap). We believe this makes sense, since we are not interested in the actual low-level location of a variable (that's why for example we compare heaps modulo the location naming), but only in its actual values and in the possible aliasing relationships with other variables. Moreover, the PHP

allocation strategy may change in future work or in different implementations, therefore we need to be agnostic in this respect.

However, this turned out to be a significant source of problems in the context of our semantics (and we believe would also be for other real-world scripting language semantics). Consider for example the program fragment

```
function foo() {
        // ... function defined here ...
}
$x = foo();
```

where the function `foo` may contain loops or conditionals, and therefore perform one or more configuration merges with the effect of leaving no guarantee about the actual low-level memory locations in which the merged variables will be stored. The problem arises here because, in the semantics of assignments, variable `$x` is evaluated to a reference, *before* the function is invoked. In particular, `$x` will evaluate to a reference `ref(L_g, "x")` where `L_g` is the memory location currently containing the array of the global variables. Let us say for now, `L_g` is actually `L_1`. Such a location is retrieved by inspecting the <globalScope> cell. Furthermore, notice that `$x` evaluates to this reference *before* the function is called and for this reason before any configuration merge is performed (if present in the function body). Now suppose that `foo` indeed performs one or more merges, which we recall consist, among other things, in the merging of the global and super global arrays. This may cause the global array to be rebuilt and stored into a different low-level location. After the function returns, say with a return value `v`, the reference `ref(L_1, "x")` will still be present as part of the remaining (partially evaluated) assignment `ref(L_1,"x") = v`. However, the heap has been rebuilt as a result of the execution of the function, and in particular the global array has been stored into another, implementation dependent location, say `L_15`. This will cause the computation of the assignment to get stuck because the current instruction is trying to assign a value to a variable "x" which is contained of an array stored in `L_1`, which is no longer present. It easy to see that other instances of this same issue could arise in other context.

One possible fix for this would have been, to systematically rename `L_1` into `L_15` everywhere

in the configuration, both in the heap but also, potentially, in other cells where this is needed, such as in the <scopes> cell. However, we found this approach impractical and computationally too expensive. Instead, we opted for a more general mechanism which essentially consists in adding one more level of indirection to our memory structure; instead of mapping variables to locations which in turn map to values, we decided to map variables to *virtual locations* which maps to heap locations which finally map to values. As we show later, this not only solves the ghost references problem, but also allows us to increase precision by enabling the distinction between *may* and *must* aliasing [76]. Moreover, this approach doesn't require the addition of any additional operation (such as renaming existing locations and modifying references) except for the generalisation of the existing mechanisms (e.g. when reading a value one now needs to take into account this additional layer, but the conceptual operation is still the same).

7.4.2 Updates to the memory model

As mentioned above, in the abstract semantics we introduce a further level of indirection in the memory model, via *virtual locations*. In practice, we augment the `<mem>` cell of the configuration by adding a new subcell `<thematrix>`:

```
<mem>
   <scopes>
      <currentScope> .K </currentScope>
      <globalScope> .K </globalScope>
      <globalStaticScope> .K </globalStaticScope>
      <superGlobalScope> .K </superGlobalScope>
   </scopes>
   <thematrix> .Map </thematrix>
   <heap> .Map </heap>
</mem>
```

holding a map $H^{\#}$ from locations to locations, so that, given a (virtual) location l', its actual content v in the heap is accessed as

$$v = H(H^{\#}(l^{'}))$$

After adding the new cell we had to adapt all the rules that previously explicitly matched heap locations, by mentioning the `<thematrix>` cell in them and updating the existing matching in the `<heap>` cell. For example, consider a simple internal rule (defined in `memory.k`) that reads the content of the memory location supplied as argument:

```
rule [zval-read-value]:
   <k> zvalRead(L,@Value) => V ... </k>
   <heap> ... L |-> zval(V,_,_,_) ... </heap>
   [internal]
```

In order to take into account the additional level of indirection, the rule above becomes

```
rule [zval-read-value]:
   <k> zvalRead(L',@Value) => V ... </k>
   <thematrix> ... L' |-> L ... </thematrix>
   <heap> ... L |-> zval(V,_,_,_) ... </heap>
   [internal]
```

where we simply renamed the input location from L to L', and added a matching `L' |-> L` in the `<theMatrix>` cell. For rules which involve writing, we apply the same idea. For example, the following rule writes a value into the input memory location

```
rule [zval-write-value]:
   <k> zvalWrite(L',@Value,V:LanguageValue) => . ... </k>
   <thematrix> ... L' |-> L ... </thematrix>
   <heap> ... L |-> zval(_=> V,_,_,_) ... </heap>
   [internal, mem]
```

Notice that we ended up introducing an internal naming convention for locations: we use L', L'_1 and so on for "virtual locations" while we keep using L, L_1, etc. for actual heap locations. This not only makes it easy and almost automatic to adapt existing rules to the new memory layout but also makes it easier for us to understand at a glance where a location is going to be matched in the rules.

After we transformed all the required rules, we ran a batch of tests to ensure everything worked in the same way as before. In fact, the presence of this further indirection should be non-observable from the point of view of program execution and it should be considered simply as an implementation strategy. However, as we'll see shortly, its presence allow us to solve the problems discussed above in an elegant way as well as to gain precision in determining the nature of aliasing (may or must).

7.4.3 Overview of the merging procedure

The $\mathbb{KPHP}$ configuration consists of around 50 cells. `mergeConfig` takes two configurations as arguments and updates the current configuration with a new configuration representing an over-approximation of the inputs. There are two broad cases in the definition of `mergeConfigs`:

- Special cases, where the merge can be performed trivially (i.e. without executing the full merging algorithms) or is *postponed*.
- Base case, where the full merging algorithms are invoked to produce a final configuration.

The special cases are discussed in this chapter on a per-need basis, in the context of the language construct or analysis feature being examined. An example of situation where `mergeConfigs` postpones the execution of the merging algorithm was given in the previous section while discussing the semantics of the `break` statement, while an example where the merging is obtained trivially is shown below where we discuss how we merge status codes.

The definition of `mergeConfigsBase`, which implements the full merging operation by applying the appropriate atomic sub-operations to merge the input configurations' sub-cells (or groups of), is shown in Figure 7.8.

7.4.4 Merging return values

Consider for example the `<returns>` cell, which contains a single abstract value overapproximating the set of values that has been returned in the current function. The accumulated return values

```
syntax K ::= "mergeConfigsBase" "(" BoxedConfig "," BoxedConfig ")"
[strict(1,2)]
rule    mergeConfigsBase(
   config(C1:Bag // remaining cells
      <returns> R1 </returns> <returns-loc> RL1 </returns-loc>
      <status> S1 </status> <mem> M1 </mem>
      <control>
         Ctr1:Bag <functionStack> St1 </functionStack>
      </control>
      <tables>
         Tables1:Bag <functions> F1 </functions>
      </tables>),
   config(C2:Bag // remaining cells
      <returns> R2 </returns> <returns-loc> RL2 </returns-loc>
      <status> S2 </status> <mem> M2 </mem>
      <control>
         Ctr2:Bag <functionStack> St2 </functionStack>
      </control>
      <tables>
         Tables2:Bag <functions> F2 </functions>
      </tables>)) =>
   updateConfig(config(
      C1 // remaining cells
      <returns> mergeReturns(R1,R2) </returns>  <returns-loc> RL1 RL2 </returns-loc>
      <status> mergeStatus(S1,S2) </status>
      <mem> mergeMem(0,M1,St1,F1,M2,St2,F2) </mem>
      <control>
         Ctr1    // remaining control cells
         <functionStack> mergeFunctionStack(St1,St2) </functionStack>
      </control>
      <tables>
         mergeTables(Tables1,Tables2)
         <functions> mergeFunctions(F1,F2) </functions>
      </tables>))
```

Figure 7.8: The $\mathbb{K}$ definition of `mergeConfigsBase`.

for the two input configurations are marked as `R1` and `R2` respectively, while in the resulting configuration the `<returns>` cell is updated with the result of computing `mergeReturns(R1,R2)`. `mergeReturns` is in fact an auxiliary procedure which, as its name implies, takes care of merging the content of the `<returns>` cell. In particular, this turns our to be as easy as computing the least upper bound of the two values `R1` and `R2`:

```
syntax  K ::= "mergeReturns" "(" K "," K ")"         [function]
rule mergeReturns(V1, V2) => lub(V1,V2)
```

7.4.5 Merging status codes

Consider instead the `<status>` cell, which contains the status code of the configuration. As mentioned above, this could be `NEXT`, to signify normal execution, `RET` to indicate the fact that a function has returned, `LOOP` in case an infinite looping behaviour was detected (see 7.3.2), `ERR` in case of a *fatal error* and `BREAK`/`CONTINUE` to indicate that the respective language constructs have been invoked. When merging two configurations, one of those having the `ERR` status code, then this erroneous configuration is discarded and the other configuration is directly returned instead. The erroneous configuration is tracked for error reporting purposes, by adding it to the special cell `errorDump`.

Finally, the status code `SKIP` is used to optimise the precise execution of conditional and loops and encodes the fact that the branch doesn't have to be considered in the merge. Consider for example a conditional

```
$x = 0;
if ($x == 1)
  $x = 1;
else
  $x = -1;
```

where in both concrete and sign domain the guard evaluates to `false`. As defined in 7.3.1, the conditional should evaluate both branches, but since in this case the `true` branch is never taken, the evaluation of that branch is skipped, and the respective status code is set to `SKIP`. When detecting this, the top-level procedure `mergeConfigs` will simply discard that branch and instead directly return the other one (corresponding to the only feasible patch). More formally, this is explained as follows. Let S be the (abstract or concrete) program state existing just before the evaluation of the conditional and G the guard of the conditional. Then, the semantics of the conditional consists in evaluating both branches in program states $filter(S, G)$ and $filter(S, \neg G)$ respectively (see 7.3.1). Since, in this case, the condition `$x == 1` always evaluate to `false`, this means that $filter($`$x == 1`$, S) = \emptyset$. However, in our collecting semantics settings, evaluating the

branch, say B_t, with $\emptyset$ as argument has no effect (i.e. $B_t(\emptyset) = \emptyset$), since no input state is provided and therefore the code fragment cannot be evaluated.

In general, given two configurations to be merged into one, the respective status codes also need to be combined into one (except for the special cases in which instead the merge is skipped or postponed), and this is done by the `mergeStatus` operation, whose definition is shown in tabular form in table 7.1.

mergeStatus	NEXT	RET	LOOP	SKIP	BREAK(n)	CONTINUE(n)	ERR
NEXT	NEXT	NEXT	NEXT	NEXT*	post	post	NEXT*
RET	NEXT	RET	?	RET*	post	post	RET*
LOOP	NEXT	?	LOOP	LOOP*	post	post	LOOP*
SKIP	NEXT*	RET*	LOOP*	N/A	BREAK(n)*	CONTINUE(n)*	SKIP*
BREAK(m)	post	post	post	BREAK(n)*	break?(n,m)	post	BREAK(m)*
CONTINUE(m)	post	post	post	CONTINUE(n)*	post	continue?(n,m)	CONTINUE(m)*
ERR	NEXT*	RET*	LOOP*	SKIP*	BREAK(n)*	CONTINUE(n)*	ERR

Table 7.1: Merging of status codes. The outcome `post` encodes the fact the the merge is not performed immediately but instead it is *post*poned for later. The `*` next to the results of combining `SKIP` with other status codes is a reminder of the fact that, for that particular cases, the merge is not performed but instead the configuration which doesn't contain `SKIP` is directly returned. For `BREAK` and `CONTINUE` the outcome depends on their integer argument as computed by the auxiliary functions `break?` and `continue?`. In particular, we have `break?(n,m)` = `break(n)` when $n = m$, and `break?(n,m)` = `post` otherwise. `continue*` is defined similarly. Finally, the combinations marked with `N/A` are never reached and therefore not considered.

7.4.6 Merging arrays and memories

The `mergeArray` algorithm is at the heart of our configuration merging mechanism. We present the algorithm in a slightly simplified manner compared the one we currently implemented in the $\mathbb{KPHP}$ abstract semantics to keep the exposition simple and, more importantly, to present the algorithm in a more general way which can be applied to other languages and/or semantic frameworks.

The `mergeArray` algorithm takes two array-memory pairs as inputs and produces a third array-memory pair as output. Since arrays can represent scopes and their properties variables, we'll use

the term "variables" to denote also array properties below. The key idea is to consider all the variables of the input arrays, and for each one determine how that variable will be represented in the output array.

If a variable exists in both arrays (i.e. in both branches of a conditional), then it will be included in the result, and its value will consist of the least upper bound of the two values associated to it in the input arrays. If a variable is defined in only one branch, then it will be included in the result as well, but its value will have to be merged with `NULL` in order to take into account the fact that it may not be defined (recall that in PHP, `NULL` is the default value returned when accessing undefined variables) in case the corresponding branch is taken. For example, in the code below

```
if (?)
  $x = 1;
else
  echo "hello";
```

the value of variable `$x` would be `1` if the true branch is taken, or `NULL` otherwise. Therefore, when merging the two branches when the truth value of the guard is unknown, the final result should map `$x` to the least upper bound of `1` and `NULL` (`Top` in some domains).

A second factor to be considered when merging two arrays is *aliasing*. If an aliasing relationship exists between two variables `$x` and `$y` in both branches, then the relationship should be preserved in the result. For example, in

```
$x = 1;
if (?)
  $y =& $x;
else
  $y =& $x;
```

it is evident that, after the conditional there will certainly be an aliasing between `$x` and `$y`. Such a situation is known as a *must* aliasing. If, on the other hand, an aliasing relationship exists in only one of the two branches, the situation is known as a *may aliasing*, as in the following example

```
$x = 1;
```

```
if (?)
  $y =& $x;
else
  $y = 1;
```

where, after the conditional, variables `$x` and `$y` *may* be aliased, depending on the chosen branch. In our framework we deal with this situation by representing, in the result, the two variables as aliased, and by assigning them the least upper bound of the original values. Notice that this is a sound choice since the end result will contain an over approximation of the possible results. If we were instead to ignore the possible aliasing relationship and keep the two variables separated, we may have ended up introducing unsoundness by not considering one of the possible behaviours.

We keep track, however, of the *nature* of the aliasing relationship, which, as mentioned, can be either *may* or *must*. In order to do this, we do not use any special flag or marker. Instead, the may or must nature of aliasing is computed as part of the `mergeArray` algorithm itself and is encoded in the *shape* of the output heap, as we'll show later with examples.

When the variables considered for merging contain scalar values, the above mechanisms are easily applied. When dealing with compound values, arrays and objects, the merging procedure needs to be called recursively so that the considered array or object is merged independently before resuming the previous merging operation. However, since there may be aliasing between the array or object being merged and other parts of the heap, the recursive call cannot be performed naively i.e. it is not really an independent call. Instead, the current aliasings and some of the information previously computed will have to be passed to the recursive call. When the recursive call returns, it provides the top level one with an updated heap (consisting of the array or object considered as an entry point and a partial heap containing the memory reachable from the array). This result is incorporated into the intermediate result of the caller and the merge operation proceeds with the next variable.

Informal examples

To clarify the overall idea, we now introduce a few informal examples containing simple conditionals, and show the result of the merging procedure. We'll also introduce a graphical notation useful to represent arrays and consequently the PHP heap. After we'll have defined the algorithm more formally, we will revisit those examples in more detail.

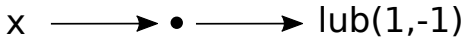
As a first example, consider the code

```
if (?)
  $x =1;
else
  $x = -1;
```

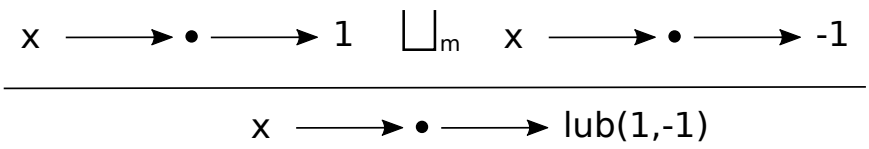
where, as usual, we denote with `(?)` the fact that the value of the conditional guard is unknown, and therefore we have to consider both branches. After the first branch is executed, variable `$x` clearly contains the value `1`. We represent this situation with the following diagram

which can be understood as follows: variable `$x` (which we recall is simply a property of the `$GLOBALS` array) is mapped to a virtual location l' (not named, but marked with a dot) which in turn is mapped to a location l, containing the value `1`. (This is a simplified representation, since we do not explicitly name locations, but it is useful when reasoning about merging heaps). The state after the second branch is executed in pseudo-parallel can then be graphically represented by the following graph

Our goal is to merge these two states into one which soundly overapproximates them. In this case this amounts to updating the value contained in `$x` with the least upper bound of the values present in the two branches:



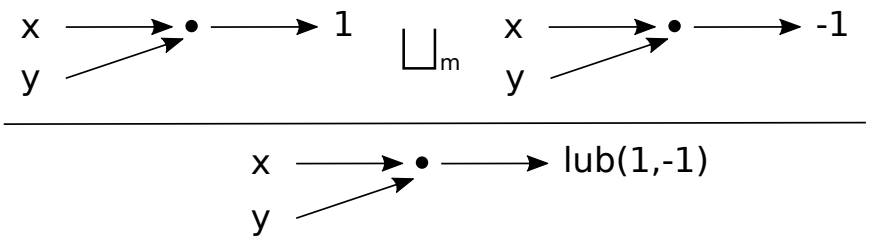
More formally, if we denote with $\sqcup_m$ the array merge operation, we can graphically represent the whole process as



Consider now the case

```
if (?)
  $x = 1;
  $y =& $x;
else
  $x = -1;
  $y =& $x;
```

where variables `$x` and `$y` are clearly aliased in both branches of the conditional (i.e. a *must* aliasing), and therefore they will definitely be aliased after the conditional has been evaluated, no matter which branch is taken. The necessary merge operation is represented as follows



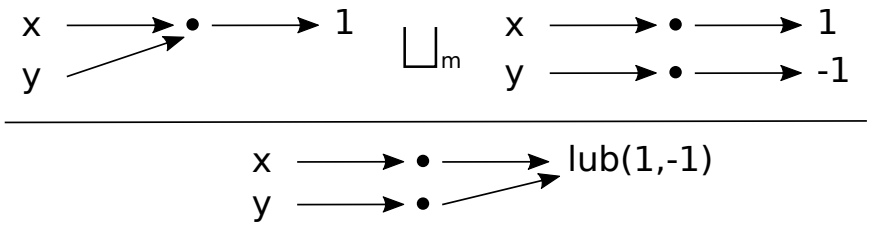
where, as usual, final values consists of the least upper bound of the original values. It is worth noticing now how we represent aliasing in the new memory model featuring the additional indirection layer and graphically represented by those diagrams. When two variables are aliased, they refer to the same memory location. The next example shows how, however, this indirection layer could be used at our advantage to distinguish between may and must alias in the result.

Consider then a program

```
$x = 1;
```

```
if (?)
  $y =& $x;
else
  $y = -1;
```

where in one of the two branches variable `$y` is aliased to `$x`, while in the other it is simply assigned the value `-1` (leading to a *may* aliasing situation). The inputs and outputs of our algorithm are summarised in the following diagram



The idea is the following. Since we are not sure wether `$x` and `$y` will be aliased after the conditional, we represent them as disjoint (i.e. non-aliased) by mapping them to different locations. However, by exploiting the additional layer of indirection, we actually map those two "virtual locations" to the same heap location, which has been updated to contain the least upper bound of all the involved values. This has the practical effect of aliasing the two variables - every lookup of either of the two will ultimately return the same heap location. However, the fact that `$x` and `$y` are mapped to two different virtual locations (which in turn are mapped to the same heap location) instead of to a single one as in the previous example tells us that the aliasing relationship between the two is that of a *may* aliasing.

Notation

We now introduce the notation necessary for an easier exposition of our algorithm. Although arrays in $\mathbb{KPHP}$ are represented as lists of triples in the form `[Key, Visibility, Location]` (where the pair consisting of key and visibility is unique), in order to discuss the algorithm we prefer to

represent arrays simply as partial functions[1]

$$e : Key \to Loc'$$

where Loc' is the set of virtual locations now replacing the usual heap locations. We will discuss later how we deal with visibility attributes. We represent the virtual heap $H^{\#}$ and the heap H, as usual, as partial functions

$$H^{\#} : Loc' \to Loc \tag{7.1}$$

$$H : Loc \to Value \tag{7.2}$$

ignoring at the moment the fact that instead of simple values we have to deal with *Zvalues*. Also, given a function $f : A \to B$, we sometimes refer to f as if it was defined as $f : \mathcal{P}(A) \to \mathcal{P}(B)$, by defining

$$f(\{x_1, x_2, \ldots x_n\}) = f(x_1) \cup f(x_2) \cup \ldots \cup f(x_n)$$

We refer to a *memory* as a pair consisting of a virtual heap and a heap

$$M = (H^{\#}, H)$$

Given two arrays e_1 and e_2 and a pair or memories $M_1 = (H_1^{\#}, H_1)$ and $M_2 = (H_2^{\#}, H_2)$, we fix the following auxiliary notation:

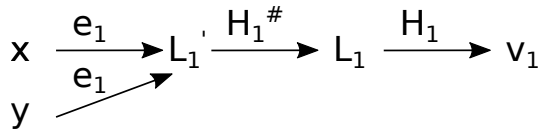
- $e_{1,2} : Key \to \mathcal{P}(Loc')$ defined as $e_{1,2}(x) = \{e_1(x)\} \cup \{e_2(x)\}$
- $pre : Loc \to Loc \to \mathcal{P}(Loc')$ defined as $pre(l_1, l_2) = H_1^{\#^{-1}}(l_1) \cup H_2^{\#^{-1}}(l_2)$

where for every function $f : A \to B$ its *preimage* f^{-1} is defined as usual as

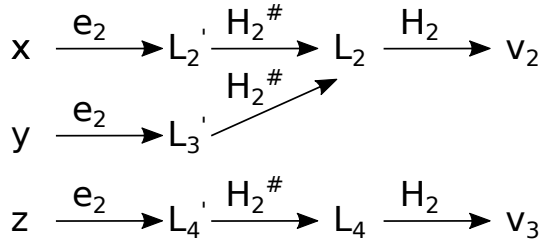
$$f^{-1}(b) = \{a \in A | f(a) = b\}$$

[1] In general, we deal with partial function by returning a special value `none` when the function is not defined on a given argument.

This concepts (as well as the examples following the discussion of the algorithm) are easily visualised with the aim of a graphical notation similar to the one we have already used to explain the algorithm informally, but more precise. Consider for example an array e_1 defining two variables x and y which are aliased. Let $M_1 = (H_1^{\#}, H_1)$ be the memory in which e_1 is defined and v_1 the value currently assigned to x and y. The situation is depicted by the following diagram



Consider now a second array e_2 (defining one additional variable z) and a second memory $M_2 = (H_2^{\#}, H_2)$ in which it is defined. In e_2, x and y are still aliased, but this time the configuration is that of a may alias (how this is obtained will become clear in the following).



We can now clarify some of the notation introduced above. For example, since we have

$$e_1(x) = l_1'$$
$$e_2(x) = l_2'$$

then

$$e_{1,2}(x) = e_1(x) \cup e_2(x) = \{l_1', l_2'\}$$

Similarly we have

$$e_{1,2}(y) = \{l_1', l_3'\}$$

and

$$e_{1,2}(z) = \{\mathsf{none}, l_4^{'}\}$$

since variable z is only defined in e_2. In this case, it is important that we record this fact by including none in the result of $e_{out}(z)$, since this information allow us to track, for example, the fact that in the final result we would have to merge the existing value of z (v_3 in the example) with NULL. Consider now variable x. Its location in the heap can be retrieved simply as $H_i^{\#}(e_i(x))$, where $i \in \{1, 2\}$. In our example, we have

$$H_1^{\#}(e_1(x)) = l_1$$

and

$$H_2^{\#}(e_2(x)) = l_2$$

meaning intuitively that in the first case x is mapped to the heap location l_1, and in the second case it is mapped to l_2. Then,

$$pre(l_1, l_2) = H_1^{\#^{-1}}(l_1) \cup H_2^{\#^{-1}}(l_2) = \{l_1^{'}\} \cup \{l_2^{'}, l_3^{'}\} = \{l_1^{'}, l_2^{'}, l_3^{'}\}$$

Notice also that, if we consider variable y instead, we would obtain the same result since x and y are aliased and therefore they share the same heap locations.

The mergeArray algorithm

Our algorithm takes the following inputs

- an input array e_1 and its associated memory $M_1 = (H_1^{\#}, H_1)$
- an input array e_2 and its associated memory $M_2 = (H_2^{\#}, H_2)$
- an output array e_{out} and its associated memory $M_{out} = (H_{out}^{\#}, H_{out})$ (initially left empty and then iteratively constructed by the algorithm)

- a map $locRemap : \mathcal{P}(Loc') \rightarrow Loc'$ keeping track of the renaming of memory locations computed during the merge. This piece of information is crucial as it keeps track, among other things, of the aliasing relationships between variables in the output memory, while it is being built. It is also essential, as will be clear from the algorithm pseudocode, in order to distinguish between may and must aliasing.

- a destination location l

and returns a memory $M = (H^{\#}, H)$ as output. The returned memory M is obtained by merging M_1 with M_2 w.r.t. the arrays e_1 and e_2. In this setting, it may help to think of M as a heap and e as an environment. Therefore, the contents of M_1 and M_2 which are reachable via e_1 and e_2 are merged. For convenience, we also define the auxiliary notation

- $H^{\#}_{out}[x] : Key \rightarrow Loc$ defined as $H^{\#}_{out}[x] \triangleq H^{\#}_{out}(e_{out}(x))$

- $H_{out}[x] : Key \rightarrow Value$ defined as $H_{out}[x] \triangleq H_{out}(H^{\#}_{out}[x])$

The complete algorithm (presented in pseudocode) is shown in Figure 7.6. We now revisit some of the informal examples that we introduced at the beginning of this section in greater detail, together with new ones.

Examples

Consider the code

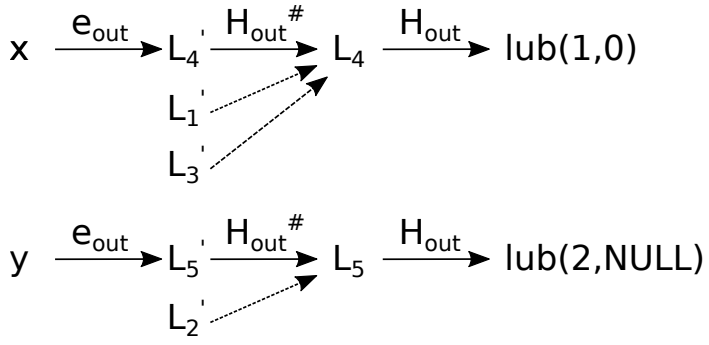
```
if (?)
  $x = 1;
  $y = 2;
else
  $x = 0;
```

the complete step-by-step execution of the merge algorithm initiated as part of the abstract evaluation of the conditional is shown in table 7.6.2. In particular, we can see a few details that were hidden in the previous simplified examples. First, notice how in the final result (reported below

START:
if if e_1 and e_2 are both empty **then**
  $H^{\#} \leftarrow H^{\#}_{out}$
  $H \leftarrow H_{out}[l \mapsto e_{out}]$
  return $M = (H^{\#}, H)$
else
  select an array element x from either e_1 or e_2
  let $l_1 = H^{\#}_1(e_1(x))$ and $l_1 = H^{\#}_2(e_2(x))$
  ▷ computing $e_{out}(x)$ and (optionally) updating *locRemap*
  if $locRemap(e_{1,2}(x)) \neq \bot$ **then**
    $e_{out}(x) = locRemap(e_{1,2}(x))$
  else
    pick a fresh (virtual) location l'
    set $locRemap(e_{1,2}(x)) = l'$
    set $e_{out}(x) = l'$
  end if
  ▷ computing $H^{\#}_{out}(x)$
  if $H^{\#}_{out}(pre(l_1, l_2)) = \emptyset$ **then**
    pick a fresh location l
    set $H^{\#}_{out}[x] = l$
  else
    set $H^{\#}_{out}[x] = l$ where $H^{\#}_{out}(pre(l_1, l_2)) = \{l\}$
  end if
  ▷ Building e_{out} and $M_{out} = (H^{\#}_{out}, H_{out})$
  set $e_{out} = e_{out}[x \mapsto e_{out}(x)]$
  ▷ Building $H^{\#}_{out}$
  $H^{\#}_{out} = H^{\#}_{out}[e_{out}(x) \mapsto H^{\#}_{out}[x]]$
  $\forall l \in pre(l_1, l_2)$ set $H^{\#}_{out} = H^{\#}_{out}[l \mapsto H^{\#}_{out}[x]]$
  ▷ Building H_{out}
  if l_1 or l_2 contains a scalar **then**
    $H_{out} = H_{out}[H^{\#}_{out}[x] \mapsto H(l_1) \sqcup H(l_2)]$
  end if
  if both l_1 and l_2 contain compound values of the same type **then**
    let $M^* = (H^{\#*}, H^*)$ the result of calling *mergeArray* with $H_1(l_1)$ and $H_2(l_2)$
    let $H^{\#}_{out} = H^{\#}_{out} H^{\#*}$
    let $H_{out} = H_{out} H^*$
  end if
  ▷ Chosen variable is merged. Cleanup phase.
  remove x from both e_1 and e_2 (if present)
  Go to START
end if

Figure 7.9: Pseudocode of the `mergeArray` algorithm.

for easier reading) although `mergeArray` has modified the original locations assigned to `$x` and `$y` (which were L_1' and L_3' respectively), those locations have been also integrated in the final result, shown with a dotted line in the diagram:



This is necessary to solve the problem of *ghost references* introduced before. Suppose the configuration merge happened in the context of a function call `foo`, and the following code had to be evaluated

```
$x = foo();
```

as explained, `$z` would be evaluated to some reference `ref(L', "x")` before the function is called and the merge takes place. However, since the merge can change the actual low-level memory location naming, this may lead to an inconsistent state where, at the point where the assignment is to be performed, the original reference `ref(L', "x")` (and in particular the location `L'`) is no longer present. Our algorithm, by keeping track of the previous locations associated to the merged variables, ensures that such inconsistent states are never encountered. The example also shows the case, for variable `$y`, where a variable (or array entry) is present in only one of the two branches. In this case, as explained before, the content of the variable is merged with `NULL`, to account for the case in which the branch where the variable isn't defined was taken at runtime.

As a second example, this time involving a *may aliasing*, consider the following program

```
if (?) {
  $x = 1;
  $y =& $x;
}
```

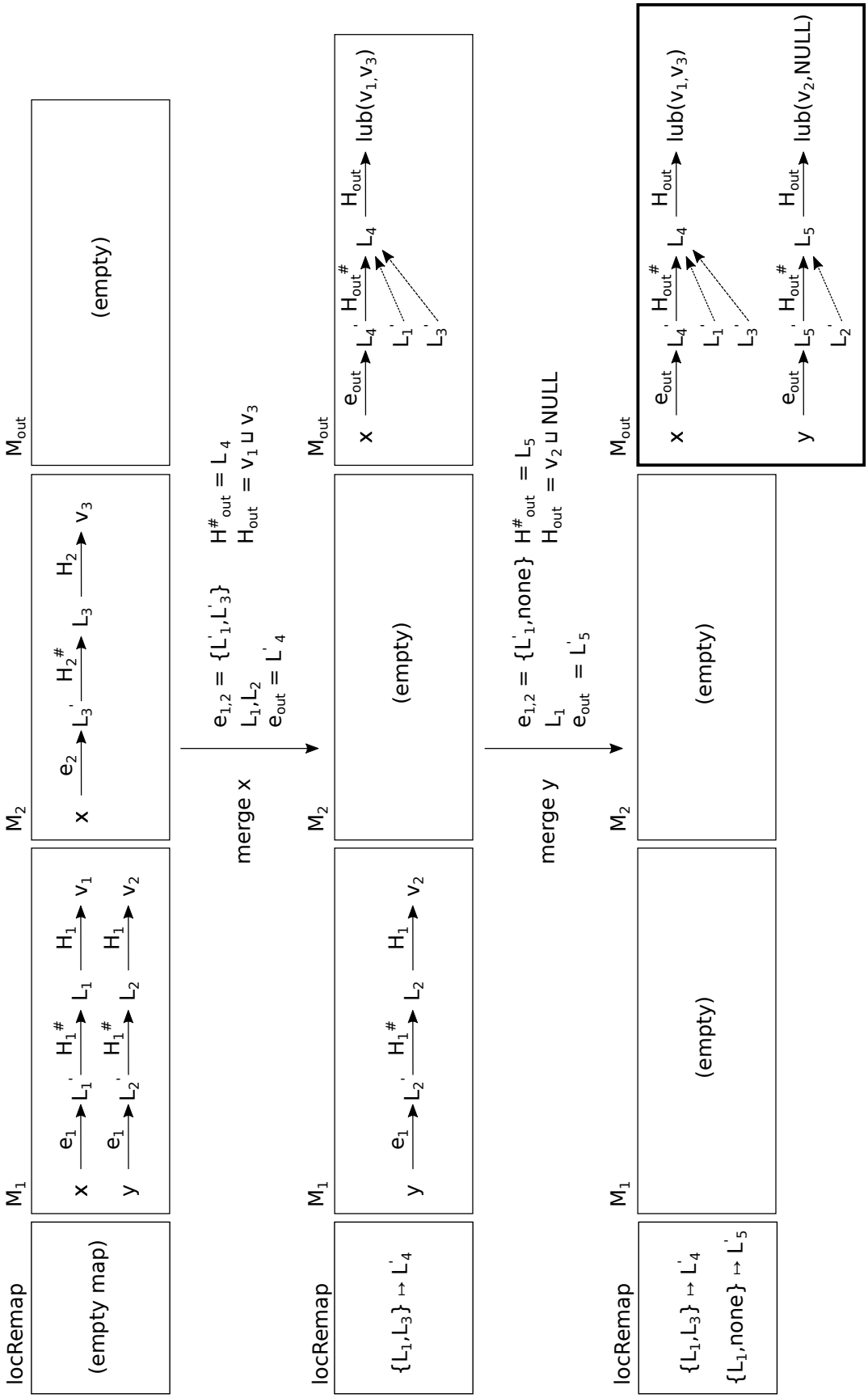


Figure 7.10: Sample execution of `mergeArray`

```
else {
  $x = 0;
  $y = -1;
}
```

Figure 7.11 shows the steps performed by `mergeArray` on this code. Notice how the final result differs from the one obtained in the previous example. As before, variables `$x` and `$y` are associated to two different (virtual) locations, L_4' and L_5'; however, those two locations are both associated to the same heap location L_4 which contains an overapproximation of the content. This particular structure when two variables are associated to different virtual location who in turn point to a same heap location is distinctive of a *may alias* and encodes the fact that `$x` and `$y` may be aliased in some execution path but may not in some other.

Finally, we consider how the algorithm deals with *must aliasing*. Figure 7.12 shows the execution of `mergeArray` on the code

```
if (?) {
  $x = 1;
  $y =& $x;
}
else {
  $x = 0;
  $y =& $x;
}
```

where, this time, variables `$x` and `$y` are mapped to the same virtual location L_4', representing a standard aliasing i.e. the same configuration that would be obtained in a concrete execution by performing a reference assignment `$x =& $y`. In fact, such an assignment is performed in both branches of the conditional, meaning that, no matter which branch is taken at runtime, `$x` and `$y` will be aliased. Once again, this situation is a *must alias*. Notice in particular the role of the map f, maintained by `mergeArray`: after `$x` has been merged, the next variable, `$y`, is considered (second row of figure 7.12). `mergeArray` then computes the set of virtual locations mapped to `$y` in both branches, which is $\{L_1'\}$ and it checks wether such a set is already present as a key in

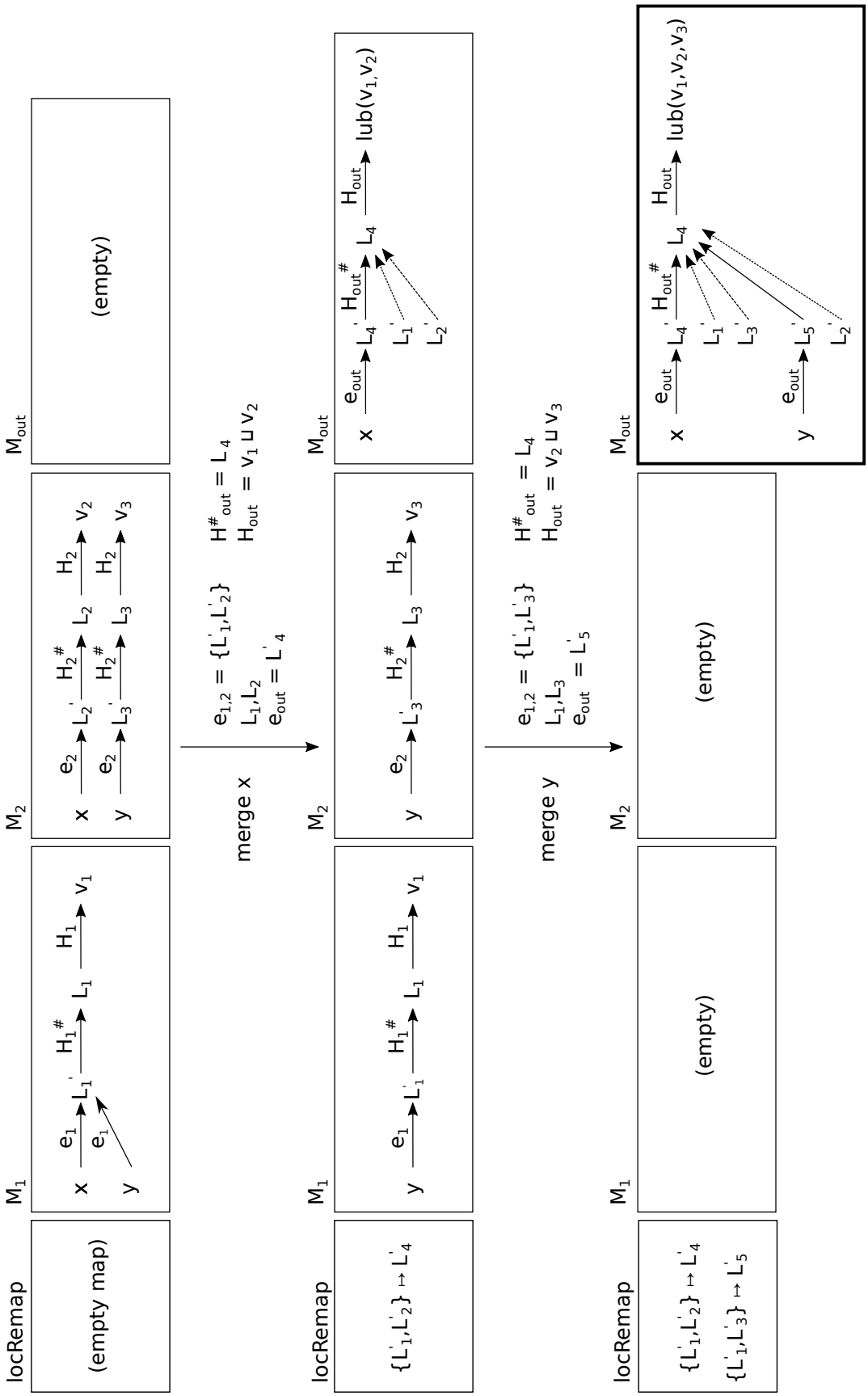


Figure 7.11: Sample execution of `mergeArray` involving a may alias

the map f. Since it it found, no further steps are necessary. Instead, `$y` is directly mapped to $f(\{L_1'\}) = L_4'$, which is the same virtual location associated to `$x`. This has the effect of producing the desired must aliasing structure between `$x` and `$y`.

Handling object visibilities

In the previous description we have deliberately omitted the details relative to the *visibility attributes* of $\mathbb{KPHP}$ array elements and object properties. Although for arrays the visibility is always `public` and therefore it can actually be ignored, for objects properties this could be either `public`, `protected` or `private` hence we have to consider cases where, during an object merge, a property has different visibility in the two branches.

This may happen only when a class itself is declared inside a conditional branch and then instantiated. In all other cases, new object properties can still be dynamically added (e.g. `$o -> foo = 1` when `foo` wasn't declared in the class declaration) but their visibility is set to `public` by default so that no mismatches can happen. Consider the following code

```
if (?) {
  class A {
        public $x = 1;
  }
  $o = new A;
}
else {
  class A {
        protected $x = -1;
  }
  $o = new A;
}
echo $o -> x; // !
```

where that the evaluation of the conditional statement entails a merge of the object associated to variable `$o`, and that its only property `$x` is defined as `public` in one branch and as `protected`

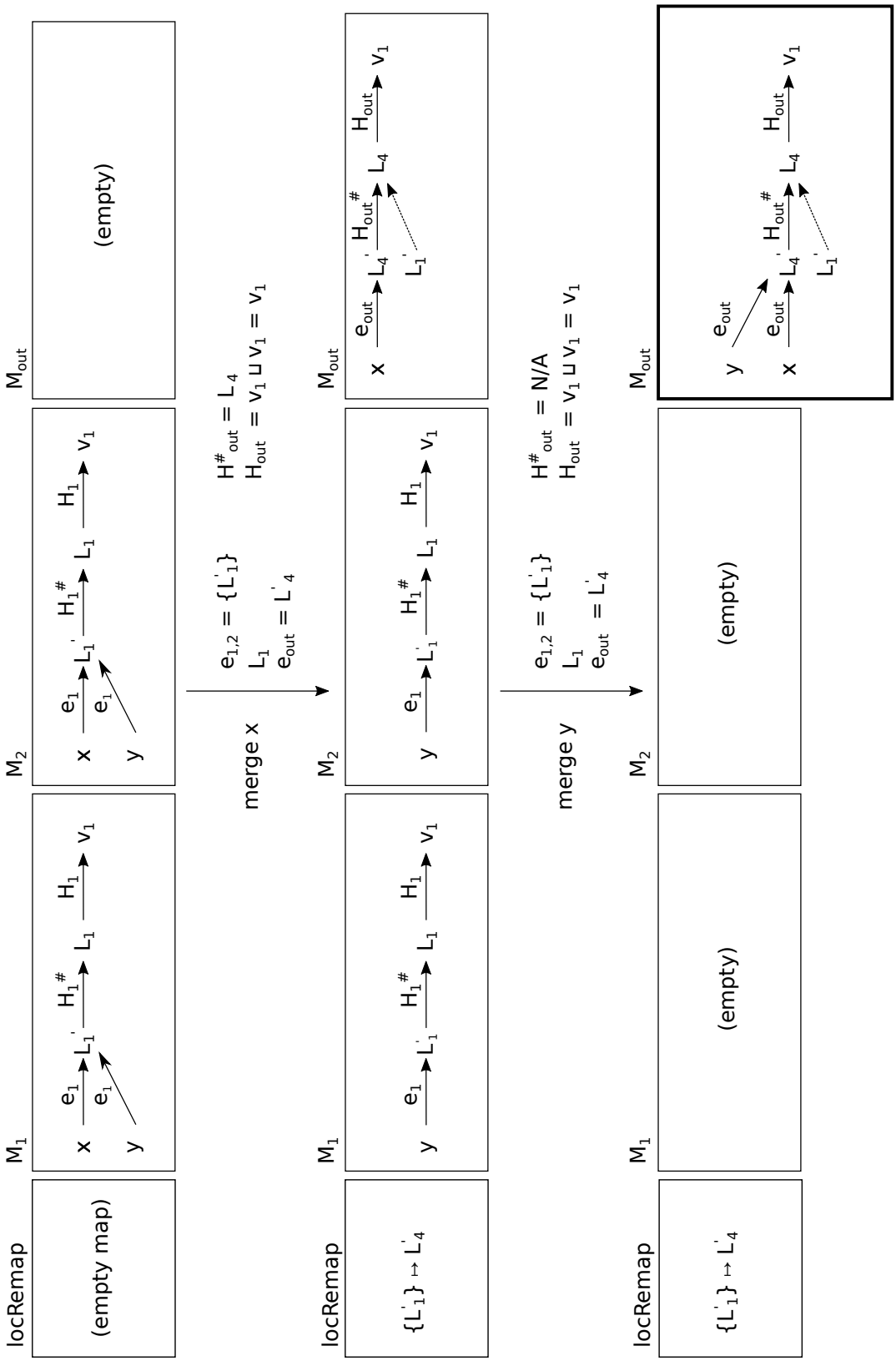


Figure 7.12: Sample execution of `mergeArray` involving a must alias

in the other. What value shall the `echo` statement in the last line output when evaluating its argument `$o -> x`? We deal with this (unlikely) case in a simple and sound way, according to the following reasoning. If the true branch was taken, then `$o -> x` would always be `public` and therefore its value would be `1`. If the false branch was taken, then `$o -> x` would be `protected`, meaning that it would be only accessible from methods of classes which are sub or super classes of `A`, while in all other cases (as in the example) the access would trigger a fatal error. Therefore, the evaluation of `$o -> x` depends on the context, and can be either `-1` or an error. Then, the set of all possible values for `$o -> x` consists of the three elements `1`, `-1` and `ERROR` - there are no other possibilities. In the result of our algorithm, the property `$x` or `$o` is simply transformed into a `public` property and assigned the least upper bound of the possible values, including error, which is represented using an internal (i.e. non accessible to PHP users) value. This ensures soundness, since the property become accessible from any context but when accessed it returns a conservative approximation of the set of all possible outcomes, including the possibility of an error when the property is accessed from a context in which it should not be visible (as in the example where `$o -> x` is called from the top-level). It can be argued that this strategy is rather imprecise, and that might be true, but as a counter argument we claim that the examined behaviour is in fact rare, and perhaps bad practice. Therefore, in the tradeoff between precision and complexity we have chosen this time to sacrifice precision in favour of decreased complexity. Indeed, we implemented this strategy as a simple addition to the `mergeArray` algorithm itself, without need to modify other semantics mechanisms such as e.g. the object lookup mechanism. In fact, doing that would be a very error prone exercise, not to mention the additional computational complexity in case we decided e.g. to modify the existing lookup strategy to deal with sets instead as with single fields.

At this point, it is worth stressing the fact that, while most existing PHP analysers do not (fully) support objects (e.g. [76, 104]), we are instead able to fully support them because in our semantics we represent objects in terms of arrays.

7.5 Array and object access, revisited

Now that we have explained how we deal with the process of merging two configurations into one and in particular the mechanisms we employ for dealing with aliasing, we can leverage them to revisit the array and object access semantics.

In PHP, array keys can be *integer* or *strings*. For objects, which are internally represented as arrays, keys can only be strings. Also, consider that array and object elements can be accessed via variables, as in `$x[$key] = 0`. Therefore, keys are in fact *first class* entities in PHP, and special care should be taken when dealing with them in an abstract domain.

A first naive solution would consist in leaving the semantics of array and object access unchanged. This would in fact work out-of-the box in our development. For example, in the sign domain, an assignment such as `$x[1] = 2` would simply become `$x[Pos] = Pos`, resulting in the creation of an array containing a single key `Pos` assigned to the value `Pos` (assuming `$x` was undefined before the assignment).

However, there are several issues with this approach. Consider for example the following program

```
$x[1] = 0;
$x[2] = 1;
1 / $x[1];
```

which leads to a division by zero (for which Zend PHP issues a warning). Suppose we now operate in the sign domain. In this setting, both assignments result in the `Pos` key to be assigned, since $\alpha(1) = \alpha(2) =$ `Pos`. If we perform the assignment as in standard $\mathbb{KPHP}$, we will assign `Zero` to the `Pos` key the first time, and then overwrite it with `Pos`, the second value assigned, meaning that after the two assignments `$x[Pos]` contains the value `Pos`, and the information about the possible division by zero is lost.

The example suggests that, in the presence of "uncertain" keys, we must instead perform a *weak update*. In other words, we cannot simply overwrite the previous value with the new one but instead we should maintain both values or an overapproximation of them. In the example,

we may want to conclude that after the two assignments `$x[Pos]` contains the least upper bound of `Zero` and `Pos`, which is simply `Top` in the sign domain. This will make sure that in the third instruction, when the division is performed, all possibilities are taken into account, including the fact that `$x[Pos]` may contain the value `0` and therefore cause an error.

However, if we were to enforce this behaviour unconditionally, we would loose flow sensitivity and the ability to execute the program in the concrete domain. Indeed, when in a concrete domain, we want to perform a *strong update*, in the sense that when assigning a variable we replace its previous value with the new one. In order to implement this strategy, we update the existing semantics of assignments.

Consider another example

```
$x[1] = 1;
$x[$k] = -1;
echo $x[1];
```

where in the second line the array `$x` is accessed via a variable `$k` which we suppose to be unknown, for example `AIntTop` in the sign domain. What happens in the last line when we attempt to read the `Pos` key? If we naively insert the new `Top` key into the array `$x` (since it was not present before), we would end up with an array containing two keys: `Pos` (assigned to `Pos`) and `Top` (assigned to `Neg`) and therefore we would read `Pos` in the last line. However, we must consider that, for example, the `Top` key represents *all* concrete keys, and therefore also the positive keys encoded by the abstract key `Pos`. In the example, we loose the information about the fact that any of those positive keys could have been updated to zero. To deal with this and similar cases, we update the reference mechanisms so that "overlapping" keys are forced to point to a same memory location via the same memory indirection mechanism we used for tracking aliasing. By doing so, we make sure that when reading or writing an abstract key, all possible affected keys are considered. We discuss this strategy in more detail in 7.5.2.

The last issue we have to deal with is related to aliasing and reference assignment. Consider again the previous example

```
$x[1] = 1;
$x[$k] = -1;
```

and suppose we successfully deal with the above problem by making sure the `Pos` and `Top` keys are forced to point to a same memory location. Suppose we now introduce an aliasing:

```
$y = 0;
$x[1] =& $y
```

If we were to perform the reference assignment as usual, i.e. by setting the key `Pos` to point to the (virtual) location containing `$y`, the `Pos` and `Top` keys of `$x` would become separated and we would loose the information about their relationship. Doing this would cause any subsequent read or write operation to any of those keys to produce an unsound state, since the information about the fact that the two keys may actually share their values is forgotten.

Therefore, when aliasing, we must take into account the fact that there might be keys which overlap with the aliasing target. In practice, we must alias all of them so that the keys relationships remains intact. To deal with this, we update the semantics of the reference assignment so that, after the operation is performed as usual, the array is rebuilt in order to take into account keys which may overlap with the target key. While doing this, their values are accumulated via the least upper bound mechanism.

7.5.1 Strong vs Weak updates

We now discuss in more detail the problem related to the update of array or object elements in the presence of abstract keys. Consider again the introductory example

```
$x[1] = 0;
$x[2] = 1;
1 / $x[1];
```

which, when operating in an abstract domain such as our example sign domain becomes

```
$x[AIntPos] = AIntZero;
$x[AIntPos] = AIntPos;
```

```
1 / $x[AIntPos];
```

Consider the first assignment, where we denote the assigned value as `V`, since its actual precise value doesn't matter for this discussion:

```
$x[AIntPos] = V;
```

Performing such an assignment is obviously an abstraction; no real program would attempt to assign to `$x[AIntPos]`. In fact, what this means is that, at runtime, the program will assign to a positive integer key of `$x`. At analysis-time, we must assume *any* of those key could be assigned to.

In this conceptual setting, we can for example think of our assignment `$x[AIntPos] = V` as encoding the *set* of possible assignments

$$\texttt{\$x[1] = v}, \texttt{\$x[2] = v}, \ldots, \texttt{\$x[MAX - 1] = v}, \texttt{\$x[MAX] = v}$$

where `MAX` denotes the maximum integer variable available in the current implementation. Consider now the effect of evaluating each assignment independently. According to the standard semantics, the first (`$x[1] = v`) would update the first element to `V` and leave the rest unchanged. Similarly, `$x[2] = V` would update the second element to `V` without changing the other elements and so on. If we were to merge the `MAX` arrays obtained by evaluating each assignment independently we would obtain as a result an array containing the same `MAX` keys, whose individual assigned values are obtained by merging the values assigned to the same key in all of the `MAX` array. For example, `$x[1]` would contain the value

$$\underbrace{\texttt{V} \sqcup \texttt{P} \sqcup \ldots \sqcup \texttt{P}}_{MAX} = \texttt{V} \sqcup \texttt{P}$$

where with `P` we denote the previous value contained assigned to the given key. Similarly, for `$x[2]` we get

$$\underbrace{\texttt{P} \sqcup \texttt{C} \sqcup \texttt{P} \sqcup \ldots \sqcup \texttt{P}}_{MAX} = \texttt{V} \sqcup \texttt{P}$$

and so on. In short, considering multiple assignments to different keys of the same array and then

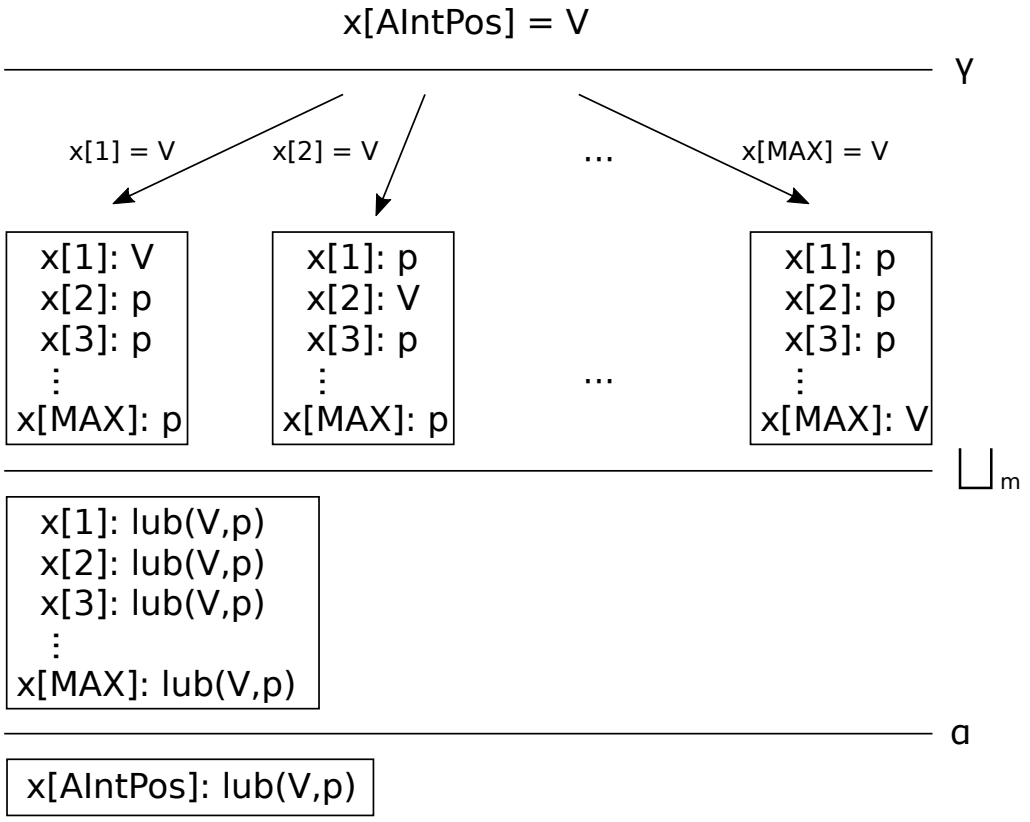


Figure 7.13: Abstract semantics of array access. A single access to an array (or object) via an abstract key can be understood as a set of pseudo-parallel accesses to all the concrete key encoded by the abstract one.

merging the results naturally leads to our intuitive idea of a weak assignment. When switching back to the abstract domain of signs, the set of keys from `1` to `MAX` gets abstracted to `AIntPos`, therefore we obtain the intended behaviour of assigning the value `V` $\sqcup$ `P` to `$x[AIntPos]`. Perhaps this is better understood via a diagram such as the one shown in figure 7.13.

On the other hand, consider assigning to a precisely identified key. This is clearly the case of a concrete domain, e.g. `$x[1] = v`, but notice that it could happen in abstract domains as well. Consider for example the assignment `$x[0] = v` in the sign domain, which becomes, after the key is abstracted, `$x[AIntZero] = v`. Since (recall section 4.5 where we defined the sign domain)

$$\gamma(\texttt{AIntZero}) = \{0\}$$

the assignment `$x[AIntZero] = v` encodes a single concrete assignment, namely `$x[0] = v`, meaning that, in our collecting semantics reasoning, the final result would simply be the result of performing this simple assignment. Once again, this is intuitively evident from figure 7.13. Since there is only one branch to be merged, the result is equivalent to the branch (assignment) itself and considering previous values is not required.

In general, the fact that an abstract key identify a unique concrete key can be formalised in terms of the cardinality of concretisation function γ. If $|\gamma(k^{\#})| = 1$, then the abstract key $k^{\#}$ identifies a unique element. Otherwise, i.e. if $|\gamma(k^{\#})| > 1$, then it identifies multiple elements. Based on the above reasoning, we update the assignment semantics so that, when assigning `$a[k] = v`, the assigned value is given by

$$a[k] = \begin{cases} a[k] \sqcup v & \text{if } |\gamma(k)| > 1 \\ v & \text{if } |\gamma(k)| = 1 \end{cases}$$

Since, in general, γ may be non computable, in our static analyser we simply define a predicate

$$isGammaSingleton : AbstractValue \rightarrow Bool$$

in every domain, returning true if an abstract value encodes a single concrete one (such as `AIntZero` above) or false otherwise (such as `AIntPos` or `AIntTop`). Clearly, in the concrete domain it will be $isGammaSingleton(x) = True$ for all values x.

7.5.2 Overlapping keys

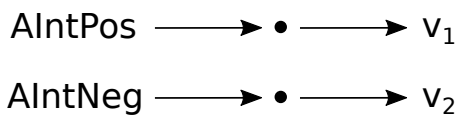
Another problem we have to deal with in the presence of abstract keys is the one of *overlapping* keys. In the introduction to this section we have already shown an example:

```
$x[1] = 1;
$x[$k] = 0;
echo $x[1];
```

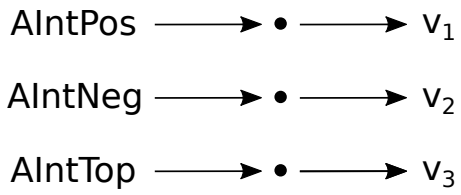
where, assuming `$k` evaluates to `AIntTop`, we would have to take into account that the other existing key `AIntPos` (as a result of the first assignment) may also need to be updated as a result of the manipulation of the key `AIntTop` in the second assignment. To better motivate this, consider another similar example:

```
$k = ?
$x[1]  = 1;
$x[-1] = -1;
$x[$k] = 0;
```

and, as usual, let's reason in the sign domain. After the second assignment `$x[-1] = -1` is performed, the array `$x` can be represented as follows



where once again we use generic values v_1 and v_2 (which, in the example, are `1` and `-1` respectively). Let us now consider the evaluation of the third and last instruction, assuming `$k` evaluates to `AIntTop`. As a first tentative solution, we could simply insert the key as we normally would in a concrete setting:



However, this strategy is clearly erroneous, as motivated before. To see this, start by considering the meaning of `AIntPos`, `AIntNeg` and `AIntTop`:

$$\gamma(\texttt{AIntPos}) = \{1, 2, \ldots\} \subset \mathbb{Z} \tag{7.3}$$

$$\gamma(\texttt{AIntNeg}) = \{\ldots, -2, -1\} \subset \mathbb{Z} \tag{7.4}$$

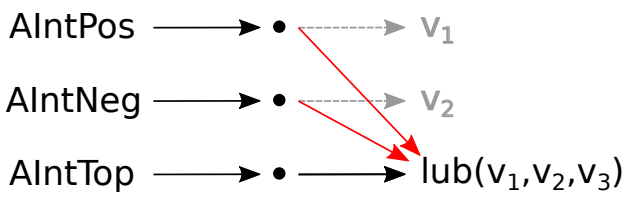
$$\gamma(\texttt{AIntTop}) = \mathbb{Z} \tag{7.5}$$

Also notice that, while

$$\gamma(\texttt{AIntPos}) \cap \gamma(\texttt{AIntNeg}) = \emptyset$$

this is not the case with `AIntTop`, since, for example $\gamma(\texttt{AIntPos}) \cap \gamma(\texttt{AIntTop}) \neq \emptyset$. This suggests that, while syntactically different in the abstract domain, the abstract keys `AIntPos` and `AIntTop` may actually refer to a same set of concrete keys, meaning that when reading or writing one of them, special care has to be taken to properly handle the other. When reading from a given key, one must consider the possible overlapped keys in order to return a correct result. When writing, one must ensure that all affected keys are updated as well.

We deal with this in a uniform and simple way, made possible by the additional layer of indirection in the memory model introduced earlier. In particular, we take advantage of the `<thematrix>` cell in order to alias overlapped array or object keys to a single memory location, which is updated to contain the least upper bound of all the affected values. Diagrammatically, the evaluation of the last assignment in the example would result in the following



where, while resolving `$[AIntTop]` to a reference, we performed the following operations

- Insert the new `AIntTop` key into the array `$x`
- Gather the set of keys $\{k_i\}$ for which $k_i \sqcap \texttt{AIntTop} \neq \bot$
- Alias those keys, via `<thematrix>` to the heap location assigned to the new key
- Update the value assigned to the `AIntTop` key just inserted, by computing the least upper bound between the target value and all affected values of the keys $\{k_i\}$

This ensures soundness both when reading and writing. When reading, the result would be a conservative approximation of the possible values. When writing, the aliasing ensures that all

possibly affected keys will be affected by the update. We implement this strategy by updating the reference resolving mechanism (originally introduced in 5.2.1). However, notice that, in general, it may not be enough to look for all the existing keys which intersects the newly inserted key, even if this was not apparent in the example above. Consider for example a more expressive domain, such as, for example, the domain of integer intervals (not available yet in our framework). Suppose an array already contains two keys `[1 3]` and `[3 5]` and we wish to insert the new key `[1 2]`. In this case, we have

$$\texttt{[1 2]} \sqcap \texttt{[1 3]} \neq \bot$$

and therefore we shall consider the key `[1 3]` for update, as explained above. However,

$$\texttt{[1 2]} \sqcap \texttt{[3 5]} = \bot$$

but since we have that

$$\texttt{[1 3]} \sqcap \texttt{[3 5]} \neq \bot$$

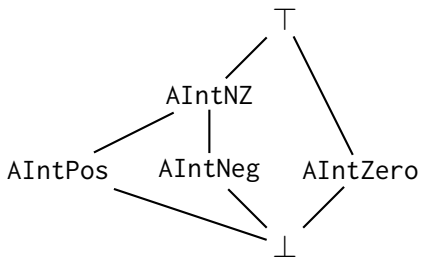
also the key `[3 5]` needs to be considered for the update. In our analysis, we include an algorithm that implement this strategy, independently from the chosen domain.

A possible, more commonly found alternative (e.g. [104, 107, 76]) would be instead to lift the reading and writing operations to sets of variables/locations. When an array element is updated, all other possibly affected keys are updated as well. When reading, the whole set of possibly involved keys is read and the least upper bound of the retrieved values is returned. Notice that, in this last case in particular, this mechanism would guarantee more precision than ours, since the reading operation would have no side-effects and it would not "collapse" the affected array elements permanently. On the other hand, our mechanism is computationally more efficient and conceptually simpler in the context of our semantics. For the efficiency, simply consider that, in our analysis, each write or read operation still involves a single lookup (instead of a set of). The cost is instead payed when inserting new keys into an array (as in the examples above), but this is done incrementally and once and for all, so that when keys are subsequently read or updated, only

a single lookup is performed, according to a semantics which closely resembles the concrete one. In this settings, the above mechanism can also be seen as a *caching* operation. Moreover, our choice is also conceptually simple, as it doesn't require extensive deep modifications to the existing reference resolving mechanism (see Section 5.2.1), which is considerably complex and constitutes a critical component of our formal semantics. Instead, the reference resolving mechanism is modified in a relatively simple way in order to enable the additional behaviour described above. This increases our confidence in the soundness of out approach and is in line with out design philosophy of attempting to generalise the existing semantics without modifying its nature ad-hoc.

Fragmented keys

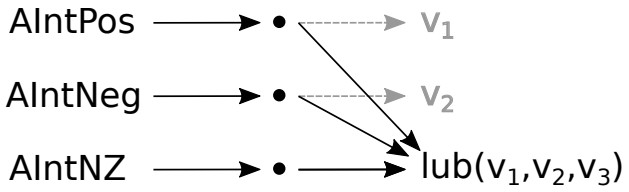
We elaborate a bit more on the problem of overlapping keys by introducing the issue of *key fragmentation*. To better illustrate the problem, we introduce a new, slightly improved sign domain, where we simply add one more element `AIntNZ` to encode all non-zero numerical values. The new lattice is shown in the following diagram



Consider now the same example as before

```
$k = ?
$x[1]  = 1;
$x[-1] = -1;
$x[$k] = 0;
```

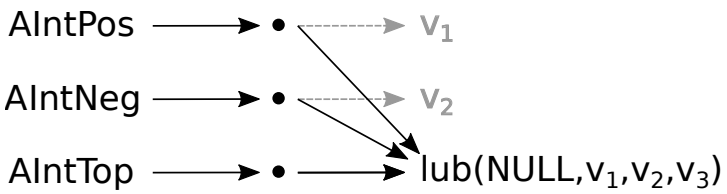
but assume now `$k` evaluates to `AIntNZ`, i.e. a non-zero integer. As discussed before the result will look like



which is indeed the correct result.

The example gives us the tools necessary to revisit the previous example, where `$k` evaluated to `AIntTop`, in order to discover a subtle issue. With the help of the graphical representation of the lattice, consider the difference: `AIntNZ` is "fully defined" by `AIntPos` and `AIntNeg` i.e.`AIntNZ` can be expressed in a unique way as `AIntPos` $\sqcup$ `AIntNeg`; on the other hand, `AIntTop` can be expressed in many ways - for example `AIntPos` $\sqcup$ `AIntZero` = `AIntNeg` $\sqcup$ `AIntZero` = `AIntTop`.

This leads to an issue with the previous strategy for dealing with overlapped keys. Returning to the case when `$k` evaluated to `AIntTop`, we shall also consider that, for example, also the concrete key `0` is encoded by `AIntTop`, but when we inserted the new key `AIntTop` into the array, the only two existing keys were `AIntPos` and `AIntNeg`. However, by inserting the new key and by associating it to a shared location containing the least upper bound of the values that were previously assigned to `AIntPos` and `AIntNeg` we are also implicitly updating the `AIntZero` key, even if it's not in the array yet. For this very reason, and since undefined variables evaluate to `NULL` by default, we shall also include `NULL` in the final value associated to the key, to take into account the possibility that the accessed key may be one which is not explicitly present in the array, as shown below



On the other hand, when `$k` evaluates to `AIntNZ` as above, there is no need to include `NULL` in the final value, since the only way to obtain `AIntNZ` is by combining `AIntPos` with `AIntNeg`, both already defined in `$x`. Therefore, there is no risk, in this case, that a concrete key in γ(`AIntNZ`) is not already initialised.

Being able to distinguish those two cases makes it possible to increase the precision of the analyser by avoiding adding `NULL` in the least upper bound when this is not necessary, as in the `AIntNZ` case above.

Our approach for dealing with array (and object) access is similar in spirit to that of Pixy [76] and Hauzar *et al.* [107] although instead of abstracting array and object keys as "first-class" values, as we do, they try to keep them concrete whenever possible (e.g. as sets of possible concrete keys). When updating an array element identified by an unknown key, those tools attempt to calculate the set of possibly affected elements and subsequently update all of them (performing a weak update). Conversely, when reading an array element identified by an unknown key, all identified possibly affected concrete elements are considered. In this respect, however, the tool of Hauzar *et al.* is more precise than Pixy. Consider for example an assignment such as `$x[$y] = 1`. In this case, Pixy does not attempt to precisely estimate possible values for `$y`. Instead, it treats `$y` as "all possible indexes" and updates all array keys accordingly, performing a weak update. On the other hand, [107] estimates the value of `$y` as a set of possible keys (or an integer interval) and only performs the update on those, so that the worst-case update of all the array elements is only performed when no better information in available about the key `$y`. While this approach might seem more precise then ours (especially in the example sign domain which we have considered until now), it has its downsides. Consider for example the case of a multi-dimensional array `$x[$y][$z]`. If both `$y` and `$z` were to evaluates to, say sets of 3 possible keys each, then the update would involve a total of 9 elements. If we added a further variable `$t`, evaluating to, say 5 possible values then the evaluation of `$x[$y][$z][$t]` would involve $3 * 3 * 5 = 45$ lookups, and so on. In our solution, however, each key always evaluate to a single abstract value, so that only a single lookup is performed, leading to a considerably lower computational complexity. This is made possible by the mechanism we introduced before when discussing the problem of *key fragmentation*, which uses the existing aliasing machinery in order to keep abstract keys possibly referring to shared concrete keys grouped together. In general however, in our framework, the decision about wether to keep array and object indexes concrete or not is left to the chosen domain. Indeed, our sample

sign domain abstracts numeric and boolean values but keeps strings concrete when possible. Other domains instead (such as our type domain) may chose to abstract strings as well, but the underlying framework is able to deal with both cases in a general way. If we were to define a domain which keeps track of e.g. sets of strings, we would obtain an analysis strategy similar to the ones just discussed. For this reason, we believe our analysis is generally more flexible, as it allow analyses of different precision, according to the chosen domain, instead of fixing a strategy for resolving accesses once and for all and with a fixed precision.

Similarly to the other approaches, PHANTM [106] also tries to keep array keys in concrete form. However, when precise information is not available, it ignores dealing with the situation altogether [106]. Note that this applies also to related constructs such as variable-variables and access to object fields identifies by a non-literal key. Finally, another related approach is that of RIPS [104]. When encountering a non-literal key, RIPS attempts to reconstruct its value by invoking a backward-directed string analysis, obtaining a set of possible values. All elements identified by the set of keys are then updated as needed. However, if the target key is identified to be user-defined, a "variable tampering" error is reported (instead of e.g. updating all array elements).

7.6 Case Studies and Evaluation

In the previous Sections, we have described the main design choices and solutions to static analysis problems that we implemented in $\mathbb{KPHP}^{\#}$, and we focused, for ease of exposition, on the *signs* and *concrete* domains. Recall, however, that since our static analyser is extensible and domain-agnostic, all the techniques we presented should work for any domain and, indeed, the development of new abstract domains is encouraged.

As a proof of concept, we developed an additional, non-trivial abstract domain for $\mathbb{KPHP}^{\#}$, aimed at performing type checking of PHP scripts in order to raise warnings when dubious constructions or implicit type conversions may be encountered. Tu further test and support our

assumption that it should be easy for third parties to develop their own abstract domains without being familiar with the whole formal semantics of PHP, the development of the type domain has been mainly performed by the Masters student *Raphaël Rieu-Helft* from *École Normale Supérieure* (Paris), during his 6-months long internship at Imperial College. This experience have shown that, even with little or no familiarity with our formal semantics of PHP, a user would be able to define his own static analyses in a relatively easy way.

In this section we summarise our results, by discussing our currently available domains (concrete, signs and types domains) and by comparing our analysis with the current state-of-the-art static analysis tools for PHP, via a series of examples and small case studies.

7.6.1 Currently Defined Abstract Domains

Currently, $\mathbb{KPHP}^{\#}$ supports three different (abstract) domains: the *concrete*, *signs* and *types* domains. As discussed before, the *concrete* domain defines the usual set of PHP values and operations and, when loaded into our framework, it causes it to replicate the original, intended PHP behaviour as specified by our concrete formal semantics $\mathbb{KPHP}$ (Chapter 5). This is achieved by defining concretisation and abstraction functions so that

$$\alpha(x) = x = \gamma(x)$$

and by providing, as domain operations, the usual operations already defined in $\mathbb{KPHP}$.

The signs domain, also discussed in the previous sections and used as a running example while illustrating the features or our analyser, abstracts numeric values into their signs, while keeping strings concrete whenever possible (otherwise, a simple "unknown string" abstract value is used). Although simple, the sign domain allowed us to develop and test our analyser in a simple framework, while still being able to observe and deal with all of the challenges of static analysis in general. Consider for example the following code snippet:

```
function foo($x) {
        $y = $x + 1;
```

```
        return($y);
}

function bar($x) {
        $y = foo($x);
        return($y * 2);
}

$z = bar(3); // $z = 8 (concrete),  $z = positive (signs)
```

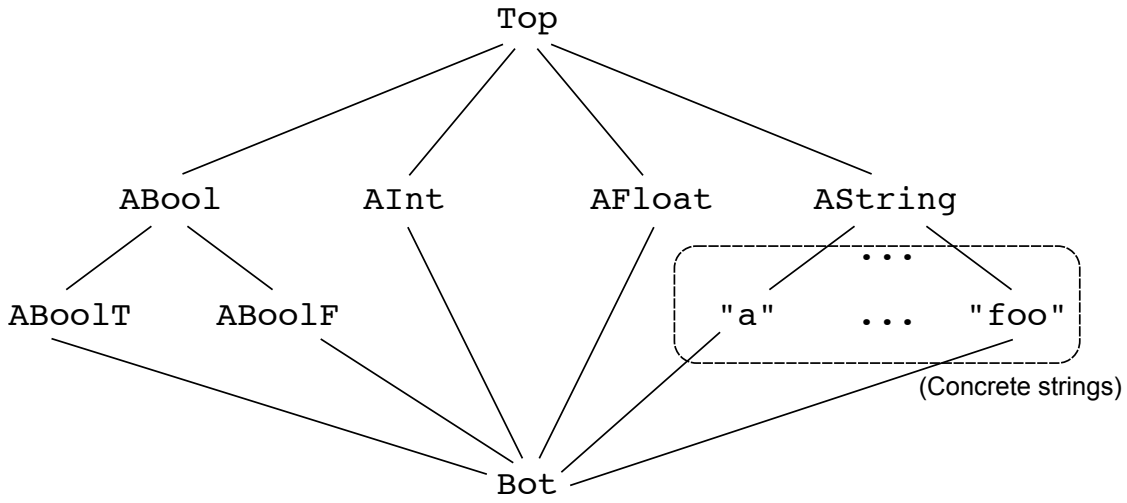
When instantiated with the signs domain, our analyser detects that variable `$z` contains a positive value. When given the concrete domain instead, the program is executed normally, and `$z` gets assigned the value `8`. One possible application of the sign domain may be, e.g. the detection of division-by-zero errors, for which PHP reports a warning.

Finally, as mentioned above, the types domain constitutes a more relevant, non-trivial example of abstract domain for our static analyser, as well as a proof of concept of the fact that our framework easily allow the definition of new analyses from scratch. We now present the types domain in some detail, before comparing our analysis framework with other tools described in related work, via a series of examples. Since the concrete and sign domains have been already discussed, we do not discuss them here in full detail. Instead, we'll provide further details and information about them during our examples and comparisons, as needed.

7.6.2 The Types Domain

The goal of the types domain is to track the possible types of variables (including arrays and object elements) during execution and report warnings when dubious constructions or type conversion may be encountered.

The types domain defines a set of *scalar values* as in the following lattice



where string values are kept concrete, unless their exact value is not statically known (in which case the value `AString`, equivalent to `AStringTop` in the sign domain, is considered). On top of this, *values* are defined as pairs consisting of a scalar value and a *list of flags*

$$Value = (ScalarValue, flag_1 \dots flag_n)$$

Flags may be used to track potentially dangerous situations such as possibly `NULL` values and implicit type conversions. For instance, the value `(AInt, MayBeNull)` denotes an integer value which may possibly be `NULL` (e.g. because it is initialised in only one branch of a conditional) while the value `(AString, ImplicitTypeConversion)` denotes a string value which has been obtained by implicitly converting a value of another type to a string. Tracking those flags together with the abstract (type) values allows the obtained analyser to point out possibly dangerous situations. Currently, the domain supports three flags: `MayBeNull`, `ImplicitTypeConversion` and `NullTypeConversion`. The last one, `NullTypeConversion`, keeps track of values which has been obtained by implicitly converting `NULL` to another type (e.g. when used in arithmetic operators, `NULL` is silently converted to `0`, or to the empty string `""` when used in a string context).

Consider the following example annotated with the computed abstract values at relevant program points

```
<?php
$y=array("foo" => "bar");
if (?)
        {$y['ind'] = "test";}
else
        echo "test"
$x= $y['ind'];
echo $x;                    // "test", MayBeNull
echo $x+1;                 // AInt, ImplicitTypeConversion
echo($x."baz");            // AStringTop, NullTypeConversion
?>
```

Since `$y['ind']` is defined in only one of the two conditional branches and the exact branch taken at runtime in statically unknown, its value *may* be `"test"` or `NULL`. Therefore, the value `("test",MayBeNull)` is assigned to `$x` after the conditional, and therefore retrieved when first accessing `$x` in `echo($x)`. In the second `echo` statement, the value `$x + 1` is computed instead. This time, since `$x` is used as an argument to an arithmetic operator, it is implicitly converted to an integer value (since the other argument, `1`, is an integer) and the analyser correctly computes the type `AInt` for the expression `$x + 1`. However, the domain has also recorded the fact that the integer value has been obtained by implicit type conversion by adding the `ImplicitTypeConversion` flag to the value. Moreover, the `MayBeNull` flag associated to the value of `$x` has been removed from the result of computing `$x + 1`, since even if the value of `$x` may be `NULL`, when converted to integer it would become `0` and therefore `1` after the addition operation - the value of `$x` cannot therefore be `NULL`. Consider now the evaluation of the last expression `$x . "baz"`. In the domain, this amount to the concatenation of the abstract values `("test", MayBeNull)` and `("baz", none)` (where `none` indicates that no additional flags are present). The concatenation operation is resolved by computing the least upper bound of the two possible results: `"testbaz"`, in case `$x` evaluated to `"test"`, or simply `"baz"` in case `$x` was `NULL` and therefore implicitly converted to the empty string during the string concatenation operation. This detail is tracked by the `NullTypeConversion` flag, which inform the user that the value might have been obtained as a result of a conversion of `NULL`

to another type. As shown in the comment on the last line of the example, the final result for the concatenation operation is the value `(AStringTop, NullTypeConversion)`, since the least upper bound of two different concrete strings is defined, in this domain, as "any string", encoded by the abstract value `AString`. A more precise, but more expensive analysis, could enable the domain to keep track of small sets of strings.

7.6.3 Discussion and Comparisons

We now discuss our development and compare its features and analysis power against some of the more recent state-of-the art tools for the static analysis of PHP code. For our comparisons we considered the tools Pixy [76], RIPS [104] and Weverca [107], also discussed in Section 4.7 and mentioned above in this Section. For Pixy, we used the latest version available at `https://github.com/oliverklee/pixy`. For Weverca, we used the online web interface at `http://perun.ms.mff.cuni.cz/weverca/` and finally, for RIPS, we used an earlier version (`http://rips-scanner.sourceforge.net`) since the latest version described in [104] wasn't yet available to the public at the time of writing.

Rationale

When comparing different static analyses, we believe we shall focus on a few different dimensions: *underlying language model*, number of *supported features*, *analysis power/techniques* and *generality of the analysis*. In general, it is desiderable that the analysis is able to deal with as many language features as possible, including the most challenging ones. On the other hand, as already discussed, analysis techniques should always be supported by a sound underlaying language model - it doesn't matter how powerful an analysis algorithm is, if the underlying assumptions about the language model are incorrect then the analysis results will be unreliable. Moreover, the usefulness of a static analysis should also be evaluated w.r.t. the range of its possible applications. In most cases, analyses are developed ad-hoc, targeting a specific application (e.g. taint analysis or type checking). While this is not a bad decision itself (it clearly depends on the specific target application), we

believe a principled design of a static analysis should lead to general-purpose analysers which can be freely configured by adding new abstract domains in order to perform different analyses, without having to redesign the entire analysis engine or language model.

Code Examples

We now show a few code examples and discuss how ours and other analyses perform on those. We shall note since now that our analyses performs considerably better than Pixy and RIPS, both in the sense of precision and supported language features. In general, our tool is able to find more bugs with less false positives (although, as discussed in 7.6.4, it cannot yet handle large real-world PHP applications). Weverca, on the other hand, considerably outperforms both Pixy and RIPS and is comparable to our work in terms of supported language features and precision, even though its underlying language model is not as precise as ours, resulting in more false positives and negatives when analysing particular features and corner cases. Pixy and RIPS are only able to detect taint-style vulnerabilities. The current implementation of Weverca also targets taint-style vulnerabilities, although the authors claim that the underlying static analysis machinery is more general and different abstract domains could be plugged into it. However, this has not been implemented/shown.

Evaluation Order. Consider again the set of examples discussed in 5.2.4 when investigating the evaluation order of the arguments of binary operators:

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


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

and

```
$a = array("one");
```

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

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

This set of examples shows that, in PHP, variables are evaluated differently from array elements when given as arguments to a binary operator. In particular, evaluation is always left-to-right, with the difference that array elements are fully evaluated to values, while variables are first both evaluated to references *before* those are actually resolved to values. This causes any side effect in the second argument to affect the final value of the first argument. Instead, when using array elements this phenomena doesn't happen, because the first argument is fully evaluated before the second's argument side effects can take place.

We slightly modified those examples in order to test wether the considered static analysis tools correctly model this behaviour. First, consider the following example:

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

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

In this case, all the tools we considered were able to detect a possible XSS vulnerability in both code snippets, as expected. However, consider the second case:

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

$a = $_POST["one"];
$c = $a . ($a = "two");
```

```
echo $c; // "twotwo"
```

In real PHP, there is a XSS vulnerability in the first snippet. However, only RIPS is able to detect it, while both Pixy and Weverca don't. This is due to an erroneous modelling of the semantics of binary operations. This example demonstrates the importance of having a solid underlying language model to support static analysis: although all the considered tools are able to support simple assignments as well as access to array elements and binary operations, the lack of precise language modelling makes them unsound (i.e. it leads to missed errors or false negatives). On the other hand, this same lack of precise modelling may also affect precision, causing the introduction of false positives. Consider the example:

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

In this case, for the same reasons above, Pixy and Weverca report a false alarm, because they do not model the fact that the first argument `$a[0]` is evaluated to a value straight away before being concatenated. Instead, they assume that at the moment of evaluating the first argument, the side effects of the second argument has already affected it. Since our tool is based on our formal semantics of PHP which correctly models the evaluation strategy discussed above, it doesn't suffer from the problem.

Variable Variables and superglobals. Among the tools we are considering, only Weverca seems to support variable variables (notice however that RIPS may support them in its latest incarnation, which is not publicly available yet). For example, given the following code fragment

```
function select($fname,$arg) {
        $result = $fname($arg);
}

function foo($x) {
        echo $x;
}
```

```
function bar($x) {
        echo $x;
}

if ($_GET["something-random"])
        $fun = "foo";
else
        $fun = "bar";

select($fun,$_GET["tainted-date"]);
```

both RIPS and Pixy are unable to detect the XSS vulnerability. Only our tool and Weverca are able to do so, since they support variable functions.

However, we were once again able to find inaccuracies in Weverca's modelling, due to its lack of a precise language model. Consider the following example:

```
function foo() {
    if ($_GET["something"])
      $v = "_GET";
    else
        $v = "z";
    $r = $$v;
    echo $r["usr"];
}
foo();
```

At a first look, it may seem that the example contains a possible XSS vulnerability. In fact, suppose the `true` branch of the conditional is taken; then, variable `$v` would contain the string `"_GET"` which, in turn, will be used to identify the name of a variable (the `$_GET` superglobal array), which is finally assigned to variable `$r`. Therefore, `$r` would be initialised with a copy of `$_GET`, whose element at index `"usr"` is finally printed, leading to a possible output of tainted data. However, a closer look at the online PHP documentation reveals that superglobals cannot be used

as variable-variables. In fact, the program above, assuming the `true` branch is taken, will *not* print the value of the superglobal variable `$_GET`, but instead it will print `NULL` , no matter what value has been previously assigned to `$_GET["usr"]`. On this example, Weverca issues a warning for a possible XSS vulnerability, which constitutes a false alarm. Since our semantics correctly model the fact that superglobals are not considered when used in variable-variables, our analyser does not suffer from this imprecision.

Implicit Type Conversions. As discussed in 3.2.2, implicit type conversions or *type juggling* are pervasive in PHP and indeed are fully modelled in our semantics. Consider the following example:

```
$x= $_GET["tainted"];
echo $x;                 // XSS
echo $x+1;       // Safe, because $x implicitly converted to Int
echo($x."baz"); // XSS
```

Here, Weverca, Pixy and our own tool are able to detect that the second `echo` statement is harmless (since the presence of an arithmetic operation causes `$x` to be implicitly converted to an integer). RIPS instead detects three vulnerabilities instead of two, because it doesn't model type juggling properly. This example, although very simple, further demonstrates the importance of basing a static analysis on a solid language model.

Tu further elaborate on the topic, consider now the following example:

```
function printName($name){
    echo $name;
}
// initialise $name to a safe string
$name = "dnf";
// initialise $x to an object
$x -> b = "a";
// this comparison is always false
if($x < "a") {
    $name=$_POST["name"];}
// $name is never assigned a tainted value, this program is safe
```

```
printName($name);
```

As highlighted in the comments, this code is safe, since the comparison `$x < "a"` is always false, and therefore the tainted value of `$_POST["name"]` is never assigned to `$name` and therefore never printed via the `echo` statement (reached when calling `printName`). The reason why the comparison is false is that the arguments of the comparison operator `<` are implicitly converted to integers; `$x`, which contains a non-empty object, is converted to `1`, while the string `"a"`, being a "non-numeric" string, is converted to `0`. However, all the tools we considered issue a false alarm, since they are not modelling implicit type conversions precisely.

Aliasing. In the literature about the static analysis of PHP, aliasing has always been considered one of the most challenging problems (see 4.7). To get started, consider the following code snippet:

```
$y["bar"]["baz"] = $_GET["foo"];
$z =& $y["bar"]["baz"];
unset($y["bar"]["foo"]);        // $z is now tainted
echo $z; // XSS!
```

RIPS fails to detect the possible vulnerability, suggesting it may have a very week support for aliasing, whereas our tool, Pixy and Weverca are able to correctly find the issue. Moreover, aliasing may also be introduced when passing arguments to functions by reference, as in the following case:

```
// untainted string
$x = "hello";
// taints the passed variable (pass by reference)
function foo(&$x) {
        $x = $_GET["taint"];
}
foo($x);
echo $x;
```

Here, variable `$x` is initially assigned an harmless value, and then it is passed to the function `foo` before being printed. The function `foo` assigns a tainted value to its argument, which is passed by reference, causing the global variable `$x` to become tainted after the call, and therefore

introducing a possible XSS vulnerability in the last line. However, only Weverca and our tool are able to detect this. Pixy and RIPS miss the error, suggesting the fact that they do not model the pass-by-reference mechanism.

Consider now a more complex example involving both aliasing and variable functions:

```
function call($fname,$arg) {
    $z[$arg] = $fname;      // function name
    $a =& $arg;
    $w =& $z[$a];           // function name in $w
    $w($a);                 // call $x($y)
}

function bar($x) {
    echo $x;
}

call("bar",$_GET["stuff"]);
```

The function named `call` takes two arguments: a function name and an argument, and call that function (using the variable function mechanism) with the provided argument. We over-complicated the function body on purpose, to test wether the considered tools were able to handle it. In particular, we first store the function name (`$fname`) into a newly initialised array `$z` at position/index given by the variable `$arg` (which also constitutes the argument to the function). Then, we alias the parameter `$arg` with a new variable `$a`. Finally, we alias a new variable `$w` with the content of `$z[$a]` (which is the same as the argument `$fname` because of the aliasing) and perform the expected payload by calling `$w($a)` (with the effect of calling `$fname($arg)`). Only our tool and Weverca, are able to handle the analysis of this program and detect the possible XSS vulnerability. Pixy and RIPS both miss the bug.

Control flow. In the following example

```
while (true) {
        echo "hi";
```

```
}
echo $_GET["foo"];
```

both Pixy and RIPS produce a false alarm by issuing an XSS warning on the last line. However, since the `while` loop doesn't terminate, no alarm should be issued because the offending `echo` statement is unreachable. In fact, our static analyser detect the situation and ignore any subsequent computation, producing a more precise result. The tool Weverca also handles this situation correctly.

Similarly, in the following example

```
$x = "hello";
while (true) {
        if ($_GET["test"])
                break;
        else {
                $x = $_GET["foo"];
                break;
                echo $x;
        }
}
```

RIPS reports a false alarm (since it is not considering the `break` statement) while all other considered tools (including ours) correctly report no alarms.

7.6.4 Evaluation

Our tool, although capable of performing deep and novel static analysis, is still a prototype and unfortunately is not yet able to analyse large codebases. This is mainly due to the following factors. First, the parser may act as a bottleneck (see 5.5) since it doesn't support some of the newest constructs that has been introduced in the latest releases of PHP. Second, the fact that our implementation relies on the $\mathbb{K}$ toolchain makes it considerably slow compared to tools developed in general purpose programming languages. This was in fact expected and we do not consider it as

a limitation of our tool/approach. The main focus on this contribution is the introduction of novel static analysis techniques for PHP as well as demonstrating the benefits of using a semantics-driven development of static analysers. Our $\mathbb{K}$ formalisation provides a formal specification of our ideas. The fact that $\mathbb{K}$ is also executable provides us with a proof-of-concept preliminary implementation for free, however this is not to be considered the final implementation of our tool. In fact, in future work, we plan to use the existing $\mathbb{K}$ specification as a starting point for the development of an industry strength static analyser, using a general purpose (and faster) programming language as well as a fully-compliant parser, together with better user interface and error messaging system. In this respect, the tools we considered for our comparison are definitely better positioned than our own.

In terms of supported language features and precision, we believe our analysis generally outperforms that of Pixy and RIPS, while it is comparable to that of Weverca. However, we have noticed that even Weverca - the most powerful among the tools we considered for our comparison - may sometimes generate a higher number of false positives/negatives, due to its lack of a proper language semantics as a foundation for the analysis (as e.g. with the order of evaluation in binary operators or the use of superglobals in variable variables). This fact demonstrates the importance of building verification tools on top of trusted language models. In this respect, our analysis is better positioned, as it is *directly* based on the language semantics.

Moreover, our static analyser is also the more generally applicable. In fact, both Pixy and RIPS are only targeting taint-style vulnerabilities. Weverca, on the other hand, claims to be more generic, but the current implementation only targets taint-style vulnerabilities. For this reason, we shall regard all those three tool as targeting a single analysis problem, while we claim our tool to be completely generic - as shown by providing three different domains for it, one of those correctly reproducing the concrete execution behaviour and therefore increasing our confidence in the trustworthiness of our algorithms.

As a general note, all the tools we surveyed in this Chapter have greatly inspired the development of our static analyser - without them and their associated literature, this work wouldn't

probably be possible.

Part IV

Conclusions & Future work

Chapter 8

Conclusions and Future Work

In Chapter 5, we introduced $\mathbb{KPHP}$, the first formal semantics for a large fragment of PHP. $\mathbb{KPHP}$ models almost all the core language (including object oriented features, aliasing, `foreach`, variable variables and functions etc.) as well as a restricted number of internal and library functions. Moreover, being written in the $\mathbb{K}$ framework, our semantics is executable and immediately and automatically yields the corresponding interpreter. The confidence in the soundness of our semantics is developed via testing, i.e. by running the automatically generated interpreter against the Zend Test Suite, the standard benchmark suite for PHP interpreters and tools. Since we were able to pass all of the tests which target the core language (i.e. we exclude tests targeting specific library functions) we claim that $\mathbb{KPHP}$ provides a faithful mathematical model of the PHP language as implemented by the Zend Engine, the de-facto reference implementation. Since no previous formal semantics of PHP was developed, we were the first to do so. However, the relevance of our work is not only limited to the PHP community. In fact, it also provides a contribution in the field of semantics engineering in general. By developing $\mathbb{KPHP}$ we have demonstrated that large-scale formalisations of real-world dynamic scripting languages, once considered a daunting task, are indeed possible (and maybe also enjoyable) when supported by the appropriate tools and methodologies. In doing so, we have investigated the use of lightweight semantics frameworks ($\mathbb{K}$) and test-driven-development methodology. $\mathbb{KPHP}$ has influenced a sequel of subsequent work such

as $\mathbb{K}$ Java [18] and $\mathbb{K}$ JS [28], among others.

In Chapter 6, we started demonstrating that formal semantics, far from being just some sort of intellectual exercise with no practical relevance, are indeed useful. To provide a first proof-of-concept of the possible applications of our semantics, we leveraged $\mathbb{K}$'s existing support for symbolic execution and LTL model checking in order to develop a simple verifier for PHP scripts. To do so, we have developed a custom LTL logics for PHP by extending standard LTL with PHP-specific atomic predicates over $\mathbb{KPHP}$ configurations. Then, we leveraged the existing tool support (provided by $\mathbb{K}$ and its Maude backend) in order to be able to verify wether a given PHP script satisfies or not one of those LTL formulae. If the formula holds, then the tool claims so; otherwise it produces a counterexample. As a practical example we have been able to analyse properties of two 3rd-party PHP functions.

In Chapter 7 we presented a more substantial application of our semantics - a general purpose, extensible and fully configurable static analyser for PHP based on our formal semantics and developed following the principles of Abstract Interpretation. We called our abstract interpreter $\mathbb{KPHP}^{\#}$, to further stress the fact that it has been systematically derived from $\mathbb{KPHP}$. $\mathbb{KPHP}^{\#}$ is itself an executable formal semantics of PHP, which has been generalised so that the underlying domain of computation is given as a parameter. Depending on the chosen domain, "execution" of a program in $\mathbb{KPHP}^{\#}$ has the effect of performing static analysis, where the property to be checked depends on the domain itself (e.g. signs, types, etc.). Furthermore, when a *concrete domain* is provided (i.e. a domain where $\alpha(x) = x = \gamma(x)$) $\mathbb{KPHP}^{\#}$ simply executes the program as in the original semantics $\mathbb{KPHP}$. This was in fact one of our design goals for $\mathbb{KPHP}^{\#}$: instead of providing ad-hoc semantics rules for "abstract behaviour", we aimed at always providing, for every relevant construct/operation, a single abstract rule which describes the desired behaviour in *any* possible domain, concrete or not. This way, the concrete semantics simply becomes an instance of the more general abstract semantics. This provides assurance about the correctness of the analysis. Since directly based on $\mathbb{KPHP}$, our abstract interpreter is able to handle most language features, including the ones that, in the literature, have been considered challenging and

therefore are sometimes not properly handled by other tools. Comparing our analysis with related work, we noticed in fact that we generally outperform other tools both in terms on correctness and precision on targeted examples. In our opinion, most of the improvements are due to the fact that our static analysis is based on a solid, trusted underlying language model. No matter how powerful analysis techniques a tool might use, if the underlying assumptions about the language behaviour are wrong, then the analysis result may be incorrect or imprecise, as we clearly demonstrated in our examples and comparisons. Modulo the need for further engineering optimisations in order to support larger codebases, we believe our approach is promising, and already favourably compares with the current state-of-the-art static analysis tools for PHP.

Overall, the main message that we aim to convey with this dissertation is the importance of basing formal methods (software verification, static analysis etc.) on a rigorous language model.

For future work, we envision the following. At the language-support level, supporting the few features that are still missing from our model: abstract classes, magic methods, interfaces and namespaces. Moreover, it would be also useful to include a more substantial set of internal/library functions in our model. Perhaps, given the recent proliferation of different PHP implementations, it may also be useful to modularise our language model so that all of those different implementations could be supported. At the static analysis level, there are several improvement from which our framework could benefit, mostly divided into two classes: usability/engineering and core analysis techniques. Some of the engineering improvements that we would definitely like to see realised are: support for larger codebases, migration to a fully compliant PHP parser, development of an online web interface and generation of better error messages. Also from the point of view of the analysis, several improvements can be done. First of all, adding support for some of the feature that are not yet supported, such as exceptions. But the most important, in our view, would be the development of several new domains such as taint-analysis, (abstract) non-interference etc., in order to further test our framework and show its wider applicability. Further tweaking/fine-tuning of the analysis engine may then be performed in parallel with the development of new abstract domains.

Bibliography

[1] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[2] N. Dershowitz. Software Horror Stories. `http://www.cs.tau.ac.il/~nachumd/horror.html`.

[3] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77*, pages 238–252, 1977.

[4] B.C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[5] R. Jhala and R. Majumdar. Software Model Checking. *ACM Compututing Surveys*, 41(4), 2009.

[6] Chucky Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012.

[7] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering*, pages 15–29, 2004.

[8] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification (CAV'07)*, pages 173–177, 2007.

[9] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th*

International Conference on Software Engineering and Formal Methods, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[11] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[12] R. Bird. *"Programs and Machines"*. "Wiley", "1976".

[13] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[14] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[15] Jacques Loeckx, Kurt Sieber, and Ryan D. Stansifer. *The Foundations of Program Verification*. John Wiley & Sons, Inc., New York, NY, USA, 1984.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[17] D. Filaretti and S Maffeis. An Executable Formal Semantics of PHP. In *ECOOP'14*, pages 567–592, 2015.

[18] Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.

[19] C. Ellison and G. Roşu. An Executable Formal Semantics of C with Applications. In *POPL '12*, pages 533–544, 2012.

[20] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javasript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014.

[21] Y. Bertot, P. Castéran, G. Huet, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development - Coq'Art - the Calculus of Inductive Constructions*. Texts in theoretical computer science. Springer, 2004.

[22] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, A. McCarthy, J, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *POPL '12*, pages 285–296, 2012.

[23] G. Roşu and T.F. Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[24] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002.

[25] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. ACM.

[26] Scott Owens. A sound semantics for ocamllight. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*, ESOP'08/ETAPS'08, pages 1–15, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Racker Development Team. The Racket Programming Language. `http://racket-lang.org`.

[28] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 428–438. ACM, June 2015.

[29] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, pages 76–87. Springer-Verlag, 2003.

[30] D. Guth. Python Semantics in K. `http://code.google.com/p/k-python-semantics/`.

[31] P. Meredith, M. Hills, and G. Roşu. A K Definition of Scheme. Technical Report Department of Computer Science UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign, 2007.

[32] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A Formal Executable Semantics of Verilog. In *MEMOCODE'10*, pages 179–188, 2010.

[33] D. Lazar. Haskell Semantics in K. `https://github.com/davidlazar/haskell-semantics`.

[34] K Team. The K Framework. `http://k-framework.org/index.php/Main_Page`.

[35] G. Roşu and T. F. Şerbănuţă. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014.

[36] International Organization for Standardization. C Language Specification - C11. ISO/IEC 9899:2011. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`, 2011.

[37] ECMA International. ECMA-262 ECMAScript Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, 2009.

[38] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[39] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ML. In *POPL '07*, pages 173–184, 2007.

[40] F. Pfenning and C. Schürmann. System description: Twelf - a meta- logical framework for deductive systems. In *CADE*, pages 202–206, 1999.

[41] M. Norrish. C Formalised in HOL. University of Cambridge Technical Report UCAM-CL-TR-453, 1998.

[42] The Compcert Team. Compcert. `http://compcert.inria.fr/`, 2013.

[43] S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43:263–288, 2009.

[44] A. W. Appel. Verified software toolchain - (invited talk). In *ESOP'11*, pages 1–17, 2011.

[45] Z. Shao and B. Ford. Advanced Development of Certified OS Kernels. University of Yale, Technical Report YALEU/DCS/TR-1436, 2010.

[46] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.

[47] J. Ŝevčik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPLì11*, 2011.

[48] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *CAV'10*, pages 258–272, 2010.

[49] D. Syme. Proving java type soundness. In *In Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[50] S. Drossopoulou and S. Eisenbach. Is the java type system sound? In *In Formal Syntax and Semantics of Java*, 1997.

[51] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of c#. In *Theoretical Computter Science (336)*, pages 235–284, 2005.

[52] Y. Gurevich. Evolving algebras. In *IFIP Congress (1)*, pages 423–427, 1994.

[53] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of java programs in javafan. In *CAV'04*, pages 501–505, 2004.

[54] M. Batty, M. Dodds, , and A. Gotsman. Library abstraction for c/c++ concurrency. In *POPL*, 2013.

[55] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T.Weber. Mathematizing c++ concurrency. In *POPL'11*, pages 55–66, 2011.

[56] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of Javascript. In *ECOOP'10*, pages 126–150, 2010.

[57] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the Full Monty. In *OOPSLA'13*, 2013.

[58] Free Software Foundation. GCC Torture Suite. `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.

[59] Arthur Charguéraud. Pretty-big-step semantics. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 41–60, Berlin, Heidelberg, 2013. Springer-Verlag.

[60] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[61] G. Roşu and A. Ştefănescu. Checking Reachability using Matching Logic. In *OOPSLA'12*, pages 555–574, 2012.

[62] A. Tozawa, M. Tatsubori, T. Onodera, and Y. Minamide. Copy-On-Write in the PHP Language. In *POPL'09*, pages 200–212, 2009.

[63] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol - http/1.1, 1999.

[64] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification, 1998.

[65] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[66] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript Web applications. In *Proceedings of CCS 2010*, pages 270–83, 2010.

[67] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[68] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[69] php.net official PHP documentation. History of PHP. `http://php.net/manual/en/history.php.php`.

[70] The PHP Group. PHP Zend Engine. `http://php.net`.

[71] HippyVM team. HippyVM. `http://hippyvm.com`.

[72] HHVM Development Team. HHVM. `http://hhvm.com`.

[73] Quercus Resin. Quercus. `http://quercus.caucho.com`.

[74] HHVM Development Team. Wikipedia on HHVM. http://hhvm.com/blog/7205/wikipedia-on-hhvm.

[75] HHVM Development Team. Announcing a specification for PHP. http://hhvm.com/blog/5723/announcing-a-specification-for-php.

[76] N. Jovanovic, C. Kruegel, and E. Kirda. Static Analysis for Detecting Taint-Style Vulnerabilities in Web Applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[77] Y. Xie and A Aiken. Static detection of Security Vulnerabilities in Scripting Languages. In *USENIX'06*, 2006.

[78] PHP Quality Assurance Team. Zend Test Suite. https://qa.php.net/write-test.php.

[79] S. Maffeis, J.C. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *APLAS '08*, pages 307–325, 2008.

[80] S. Maffeis and A. Taly. Language-Based Isolation of Untrusted JavaScript. In *CSF'09*, pages 77–91, 2009.

[81] S. Maffeis, J.C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS'2009*, pages 505–522, 2009.

[82] php.net official PHP documentation. Reference Counting Basics. http://php.net/manual/en/features.gc.refcounting-basics.php.

[83] Paul Biggar. Critiquing Facebook?s new PHP spec. http://blog.circleci.com/critiquing-facebooks-new-php-spec/.

[84] NIST National Institute of Standards and Technology. National Vulnerability Database, CVE and CCE Statistics Query Page. http://web.nvd.nist.gov/view/vuln/statistics.

[85] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet Denial of Service: Attack and Defense Mechanisms (Radia Perlman Computer Networking and Security)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[86] Open Web Application Security Project. OWASP Top 10. 2013. http://www.owasp.org/index.php/Top_10_2013.

[87] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011.

[88] OWASP. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet - Checking The Referer Header. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet#Checking_The_Referer_Header.

[89] Samy Kamkar. Technical Explanation of the MySpace Worm . http://namb.la/popular/tech.html.

[90] MySpace. MySpace. https://myspace.com.

[91] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07*, 2007.

[92] xkcd. Exploits of a Mom. http://xkcd.com/327/.

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

[94] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. 1999.

[95] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[96] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[97] P. Godefroid, N. Klarlund, and S. Koushik. Dart: Directed automated random testing. In *In Programming Language Design and Implementation (PLDI*, 2005.

[98] E. Bouwers. PHP-Sat. `http://www.program-transformation.org/PHP/PhpSat`.

[99] Fortify Team. Fortify Code Secure. `http://www.armorize.com/codesecure/`.

[100] Y. Minamide. Static approximation of dynamically generated Web pages. In *WWW'05*, 2005.

[101] www.W3C.org. HTML 4.01 Specification. `http://www.w3.org/TR/html401/`.

[102] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE'08*, 2008.

[103] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of php feature usage: A static analysis perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 325–335, New York, NY, USA, 2013. ACM.

[104] Johannes Dahse and Thorsten Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *NDSS Symposium 2014*, pages 23–26, 2014.

[105] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 989–1003, Berkeley, CA, USA, 2014. USENIX Association.

[106] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Phantm: PHP analyzer for type mismatch. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 373–374, 2010.

[107] David Hauzar and Jan Kofron. Framework for static analysis of PHP applications. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 689–711, 2015.

[108] Mark Hills, Paul Klint, and Jurgen J. Vinju. Static, lightweight includes resolution for php. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 503–514, New York, NY, USA, 2014. ACM.

[109] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF - Reference Manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[110] D. Bogdănaş. Label-Based Programming Language Semantics in the K Framework with SDF. In *SYNASC'12*, pages 170–177, 2012.

[111] PHP internals book. Memory management. `http://www.phpinternalsbook.com/zvals/memory_management.html`.

[112] PHP online documentation. Introduction to variables. `http://php.net/manual/en/internals2.variables.intro.php`.

[113] PHP Bug Tracking System. Using internal hash position is tricky. `https://bugs.php.net/bug.php?id=16227`.

[114] PHP online documentation. User-defined functions. `http://php.net/manual/en/functions.user-defined.php`.

[115] PHP Bug Tracking System. Assignment changes order of evaluation of binop expression. `https://bugs.php.net/bug.php?id=61188`.

[116] Tianchen Wei. Development of a Graphical Debugger for PHP 5. MSc final year project, Imperial College London, 2014.

[117] Shijiao Yuwen. Design and Implementation of a Static Analysis tool for PHP using the $\mathbb{K}$ framework. MSc final year project, Imperial College London, 2013.

[118] A. Arusoaie, D. Lucanu, and V. Rusu. A Generic Framework for Symbolic Execution. In *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 281–301. Springer International Publishing, 2013.

[119] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude LTL model checker and its implementation. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, pages 230–234, 2003.

[120] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[121] T.Y.C. Woo and M. Lam. A Semantic Model for Authentication Protocols. In *Security and Privacy (SP)*, pages 178–194, 1993.

[122] phpMyAdmin Team. phpMyAdmin. `http://www.phpmyadmin.net/home_page/index.php`.

[123] PHP Documentation. Cryptographic Function pbkdf2. `http://php.net/manual/en/function.hash-hmac.php`.

[124] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *WWW'04*, 2004.

[125] Sooel Son and Vitaly Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, pages 8:1–8:13, New York, NY, USA, 2011. ACM.

[126] Patrick Camphuijsen, Jurriaan Hage, and Stefan Holdermans. Soft typing PHP with PHPvalidator. Technical Report, Utrecht University, 2009.

[127] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[128] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

Part V

Appendixes

Appendix A

Complete source code of the LTL examples

A.1 Input validation function from PHP My Admin

```
/**
 * checks given $var against $type or $compare
 *
 * $type can be:
 * - false       : no type checking
 * - 'scalar'    : whether type of $var is integer, float, string or boolean
 * - 'numeric'   : whether type of $var is any number repesentation
 * - 'length'    : whether type of $var is scalar with a string length > 0
 * - 'similar'   : whether type of $var is similar to type of $compare
 * - 'equal'     : whether type of $var is identical to type of $compare
 * - 'identical' : whether $var is identical to $compare, not only the type!
 * - or any other valid PHP variable type
 *
 * <code>
 * // $_REQUEST['doit'] = true;
 * PMA_isValid($_REQUEST['doit'], 'identical', 'true'); // false
 * // $_REQUEST['doit'] = 'true';
 * PMA_isValid($_REQUEST['doit'], 'identical', 'true'); // true
```

```
 * </code>
 *
 * NOTE: call-by-reference is used to not get NOTICE on undefined vars,
 * but the var is not altered inside this function, also after checking a var
 * this var exists nut is not set, example:
 * <code>
 * // $var is not set
 * isset($var); // false
 * functionCallByReference($var); // false
 * isset($var); // true
 * functionCallByReference($var); // true
 * </code>
 *
 * to avoid this we set this var to null if not isset
 *
 * @param mixed &$var    variable to check
 * @param mixed $type    var type or array of valid values to check against $var
 * @param mixed $compare var to compare with $var
 *
 * @return boolean whether valid or not
 *
 * @todo add some more var types like hex, bin, ...?
 * @see     http://php.net/gettype
 */

function PMA_isValid(&$var, $type = 'length', $compare = null)
{

    if (! isset($var)) {
        // var is not even set
        return false;
    }
```

```
    if ($type === false) {
        // no vartype requested
        return true;
    }

    if (is_array($type)) {
        return in_array($var, $type);
    }

    // allow some aliaes of var types
    $type = strtolower($type);
    switch ($type) {
    case 'identic' :
        $type = 'identical';
        break;
    case 'len' :
        $type = 'length';
        break;
    case 'bool' :
        $type = 'boolean';
        break;
    case 'float' :
        $type = 'double';
        break;
    case 'int' :
        $type = 'integer';
        break;
    case 'null' :
        $type = 'NULL';
        break;
    }

    if ($type === 'identical') {
```

```
        return $var === $compare;
    }

    // whether we should check against given $compare
    if ($type === 'similar') {
        switch (gettype($compare)) {
        case 'string':
        case 'boolean':
            $type = 'scalar';
            break;
        case 'integer':
        case 'double':
            $type = 'numeric';
            break;
        default:
            $type = gettype($compare);
        }
    } elseif ($type === 'equal') {
        $type = gettype($compare);
    }

    // do the check
    if ($type === 'length' || $type === 'scalar') {
        $is_scalar = is_scalar($var);
        if ($is_scalar && $type === 'length') {
            return (bool) strlen($var);
        }
        return $is_scalar;
    }

    if ($type === 'numeric') {
        return is_numeric($var);
    }
```

```
    if (gettype($var) === $type) {
        return true;
    }

    return false;
}
```

A.2 Password hashing

```
/*
 * PBKDF2 key derivation function as defined by RSA's PKCS #5:
 * https://www.ietf.org/rfc/rfc2898.txt
 * $algorithm - The hash algorithm to use. Recommended: SHA256
 * $password - The password.
 * $salt - A salt that is unique to the password.
 * $count - Iteration count. Higher is better, but slower.
 * Recommended: At least 1024.
 * $key_length - The length of the derived key in bytes.
 * $raw_output - If true, the key is returned in raw binary format.
 * Hex encoded otherwise.
 * Returns: A $key_length-byte key derived from the password and salt
 * Test vectors can be found here: https://www.ietf.org/rfc/rfc6070.txt
 * This implementation of PBKDF2 was originally created by defuse.ca
 * With improvements by variations-of-shadow.com
 */

function pbkdf2($algorithm, $password, $salt, $count,
                $key_length, $raw_output = false)
{
    $algorithm = strtolower($algorithm);
    if(!in_array($algorithm, hash_algos(), true))
```

```
        die('PBKDF2 ERROR: Invalid hash algorithm.');
    if($count <= 0 || $key_length <= 0)
        die('PBKDF2 ERROR: Invalid parameters.');

    $hash_length = strlen(hash($algorithm, "", true));
    $block_count = ceil($key_length / $hash_length);

    $output = "";
    for($i = 1; $i <= $block_count; $i++) {
        // $i encoded as 4 bytes, big endian.
        $last = $salt . pack("N", $i);
        // first iteration
        $last = $xorsum = hash_hmac($algorithm,
                        $last, $password, true);
        // perform the other $count - 1 iterations
        for ($j = 1; $j < $count; $j++) {
            $xorsum ^= ($last = hash_hmac($algorithm,
            $last, $password, true));
        }
        $output .= $xorsum;
    }

    if($raw_output)
        return substr($output, 0, $key_length);
    else
        return bin2hex(substr($output, 0, $key_length));
}
```

Auxiliary functions. Below, we show the simplified implementation of the functions invoked by pbkdf2.

```
// approximation of 'strtolower': returns the string unchanged
function strtolower($s)
```

```
{
        return $s;
}

// check wether element $x is in $array
function in_array($x, $array)
{
        foreach ($array as $elem)
        {
                if ($x == $elem)
                        return true;
        }
        return false;
}

// emulates PHP hash_algos() built-in function
function hash_algos()
{
        $x = array(
                "md2",
                "md4",
                "md5",
                "sha1",
                "sha224",
                "sha256",
                "sha384",
                "sha512",
                "ripemd128",
                "ripemd160",
                "ripemd256",
                "ripemd320",
                "whirlpool",
                "tiger128,3",
```

```
                "tiger160,3",
                "tiger192,3",
                "tiger128,4",
                "tiger160,4",
                "tiger192,4",
                "snefru",
                "snefru256",
                "gost",
                "adler32",
                "crc32",
                "crc32b",
                "salsa10",
                "salsa20",
                "haval128,3",
                "haval160,3",
                "haval192,3",
                "haval224,3",
                "haval256,3",
                "haval128,4",
                "haval160,4",
                "haval192,4",
                "haval224,4",
                "haval256,4",
                "haval128,5",
                "haval160,5",
                "haval192,5",
                "haval224,5",
                "haval256,5");
        return $x;
}

// returns an array containing the expected output for the
// various hashing algorithms
```

```
function hash_algos_len()
{
        $x = array(
                "md2" => 32,
                "md4" => 32,
                "md5" => 32,
                "sha1" => 40,
                "sha224" => 56,
                "sha256" => 64,
                "sha384" => 96,
                "sha512" => 128,
                "ripemd128" => 32,
                "ripemd160" => 40,
                "ripemd256" => 64,
                "ripemd320" => 80,
                "whirlpool" => 128,
                "tiger128,3" => 32,
                "tiger160,3" => 40,
                "tiger192,3" => 48,
                "tiger128,4" => 32,
                "tiger160,4" => 40,
                "tiger192,4" => 48,
                "snefru" => 64,
                "snefru256" => 64,
                "gost" => 64,
                "adler32" => 8,
                "crc32" => 8,
                "crc32b" => 8,
                "salsa10" => 128,
                "salsa20" => 128,
                "haval128,3" => 32,
                "haval160,3" => 40,
                "haval192,3" => 48,
```

```
                "haval224,3" => 56,
                "haval256,3" => 64,
                "haval128,4" => 32,
                "haval160,4" => 40,
                "haval192,4" => 48,
                "haval224,4" => 56,
                "haval256,4" => 64,
                "haval128,5" => 32,
                "haval160,5" => 40,
                "haval192,5" => 48,
                "haval224,5" => 56,
                "haval256,5" => 64);
        return $x;
}

// simple ceiling function
function ceil ($x)
{
        $x_int = (int) $x;

        if ($x === 0)
                return 0;
        if ($x == $x_int)
                return $x_int;
        else
                return $x_int + 1;

        return 123;
}

// approximate 'hash' function - only preserves expected output length
// and type (which is string)
function hash($algo, $pass, $raw = false)
```

```
{
        $algos_len = hash_algos_len();
        $exp_len = $algos_len[$algo];
        $out = "";
        for ($i = 0; $i < $exp_len; $i++)
        {
                $out .= "*";
        }
        return $out;
}

// approximate 'hash_hmac' function - actually uses 'hash' (see above)
function hash_hmac($algo, $salt, $pass, $raw = false)
{
        $result = hash($algo, $pass, $raw);
        return $result;
}

function bin2hex($x) {return $x;}

function pack($format, $x) {return $x;}
```