Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

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

www.elsevier.com/locate/entcs

Abstract Domains for Type Juggling

Vincenzo Arceri[1]

Department of Computer Science, University of Verona, Italy

Sergio Maffeis[2]

Department of Computing, Imperial College London, UK

Abstract

Web scripting languages, such as PHP and JavaScript, provide a wide range of dynamic features that make them both flexible and error-prone. In order to prevent bugs in web applications, there is a sore need for powerful static analysis tools. In this paper, we investigate how Abstract Interpretation may be leveraged to provide a precise value analysis providing rich typing information that can be a useful component for such tools.
In particular, we define the formal semantics for a core of PHP that illustrates *type juggling*, the implicit type conversions typical of PHP, and investigate the design of abstract domains and operations that, while still scalable, are expressive enough to cope with type juggling. We believe that our approach can also be applied to other languages with implicit type conversions.

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

1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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