Combining languages using metaprogramming — PScript

University of Turku
Department of Computing
Master of Science Thesis
Computer science
July 2024
Lassi Haapala

UNIVERSITY OF TURKU
Department of Computing

Lassi Haapala: Combining languages using metaprogramming — PScript

Master of Science Thesis, 76 p.
Computer science
July 2024

This thesis introduces a novel multistage preprocessor solution, PScript. PScript is built to improve upon the existing server-to-client relationship between PHP and JavaScript. These improvements are presented through standardised design goals and code-level PScript implementations. The improvements are achieved through the PScript preprocessor, which introduces a set of additional features into the existing multistage environment. These features include direct improvements such as scoping of existing JavaScript code and hygienic variable transfers between PHP and JavaScript. Additionally PScript provides a set of features based on the concepts of metaprogramming and multistage languages, e.g., conditional compilation and expression injection. Ultimately, the thesis argues that through these features PScript is able to improve upon the PHP-JavaScript relationship both in usability, efficiency and clarity.

Keywords: metaprogramming, multistage-language, preprocessor, PHP, JavaScript

Contents

1 Introduction

Metaprogramming [1] refers to the development of software programs that are capable of treating other programs as data. Meta-level programs, referred to as metaprograms, have the ability modify an existing child-program programmatically. This ability enables programs, such as high level language compilers [2] to optimise, abstract and even generate completely new source code for a designated target program.

Metaprogramming as an industry tool has developed into a commonly used but often resented methodology. On the one hand, metaprogramming provides developers with both powerful and high level control over the compilation and execution of a designated target program. On the other hand, the resulting source code is often deemed unreadable or too volatile to be used in a standardised professional environment.

Despite the resentment, metaprogramming has made its way into commonly used tools throughout the industry. Tools such as code analyzers, linters and optimizers are a core part of modern software development. All of said tools rely on the power of metaprogramming to deliver efficiency, standardisation and code readability to their respective environments.

This thesis delves deeper into the above duality of metaprogramming and introduces a novel metaprogramming solution in the form of a language preprocessor — PScript [2]. The presented preprocessor solution employs the power of metapro-

gramming in a secure manner. The end-product is designed to meet the industry standards, while pushing for the inclusion of additional metaprogramming features.

The thesis hones in on the existing language relationship between PHP [3] and JavaScript [4] and argues that the relationship can be enhanced by employing a novel preprocessor solution. The argument is based on the existing metaprogrammability of client-side JavaScript code from a server-side PHP host. Much like a designated metaprogram the PHP host is able to analyze and modify any deployed client-side JavaScript code but much like other metaprogramming solutions, this ability remains underemployed.

Consequently, the thesis presents PScript, a metaprogramming solution that enhances the PHP-JavaScript relationship with additional security and convenience. Concurrently, the thesis presents a novel language preprocessor, PScript, that automatizes the security and employment of metaprogramming within the existing language relationship. The presented preprocessor solution is shown to improve the metaprogrammability of the language relationship, through the introduction of a novel pseudo runtime inside the PHP host.

The thesis demonstrates PScript's usability through code-level examples produced through a standardized PHP host server and a Chromium based client browser. Each example is designed to meet different industry standards in both usability, efficiency and security. Based on the above design goals, the thesis demonstrates that PScript can be included in any standardized PHP project to deliver immediate improvements.

Concurrently the thesis introduces the concept of multistage languages [5]. Multistage languages cover the relationship between multiple programming languages by considering them as stages of one meta-level language. Through the introduction of a novel PScript runtime, the thesis demonstrates how the PHP-JavaScript relationship can be considered as a singular multistage language. Furthermore, the

thesis demonstrates how the created multistage language can be used to introduce a set of novel metaprogramming features into the PHP-JavaScript relationship. [1], [2], [5]–[10]

2 Background

This thesis seeks to improve upon an existing meta-level relationship between PHP [3] and JavaScript [4]. Both languages have been popularised throughout the years, especially in web development. Together they dominate the majority of web based applications and, therefore, share a common place in the modern web development landscape.

PHP and JavaScript commonly interact in PHP-based web applications. PHP acts as the server-side rendered scripting language that makes up the core of an application, while JavaScript handles client-side interactions between the browser and the PHP-based server interface. PHP is, therefore, responsible for providing the client-side browser with pre-compiled JavaScript source code. The JavaScript source code is, in turn, modifiable programmatically via the PHP runtime. The modifications can occur during the deployment of the JavaScript payload, during which the lexical source code is exposed to the PHP runtime. [3], [4]

2.1 PHP: Hypertext Preprocessor

PHP: Hypertext Preprocessor (PHP) [3] is an universal scripting language originally developed for the purpose of simplifying web development. PHP, written in C, was developed in 1994 as a Common Gateway Interface binary for the purpose of simply tracking site visitations. After its original purpose was fulfilled, PHP was open sourced and the language's development was taken towards a general purpose

programming and scripting language. The version of PHP popular today took heavy inspiration from C, PERL and similar languages of the time, in order to create an effective toolkit and scripting language for web development.

Modern PHP is maintained by the PHP foundation [3] and the language has become one of the most popular languages in web development. PHP, as the name entails, is ultimately a scripting language with the ability to process and deliver server-side generated HTML and JavaScript code. The modern PHP, however, has grown to support the many paradigms expected of a modern general purpose programming language. These include programming paradigms such as object oriented programming, functional programming and metaprogramming.

The core PHP language has acknowledged and implemented prominent metaprogramming features that become key considerations when implementing a new solution into the PHP environment. For instance, one of these metaprogramming features is PHP's ability to evaluate code dynamically during runtime. PHP's `eval` function can digest a string of PHP code. It then proceeds to evaluate this string as PHP code, including syntax validation. The `eval` function therefore allows for dynamic code generation via lexical manipulation of a source code string. The execution of an `eval` function is also able to pass variables into a separate context where the lexical statement of the new variable is evaluated.

Additionally PHP supports a host of interactive functions that allow for the modification and observation of runtime objects. For instance the `invoke`, `get`, and `set` functions belong to a set of magic functions that can interact with objects during runtime. With the `get` function, object attributes can be analysed and reflected against, allowing for dynamic meta-level behaviours during runtime. The `set` magic function allows for the addition and generation of new object attributes at runtime. The `invoke` function blurs the lines between objects and functions, allowing objects to be called as if they were functions. [3], [11]

2.2 JavaScript

Originally developed in the late nineteens for the growing user-base of the newly founded world-wide-web, JavaScript [4] remains one of the core languages of modern web development. During the time of its creation JavaScript was meant to bring interactive functionality to so far statically served web pages of the www-era. The language was, therefore, designed to be an approachable piece of technology, with the ability to bring the functionality of the web to the masses. JavaScript was developed with a high-level language syntax that would enable interacting with the web technologies of the time, i.e., HTML and CSS.

After its initial development, JavaScript has seen many changes, but the purpose of the language has remained more or less the same. A great majority of modern web applications employ client side JavaScript to allow for effective user interaction inside a client-side browser. Together with the aforementioned standards of HTML and CSS, JavaScript, therefore, enables the modern web to be interactive and user friendly.

In the late nineteens JavaScript was far from being a standardised language. During its popularisation JavaScript was brought to ECMA international [4] to achieve a unified standard for the growing language. The developed ECMAScript standard [4] would become the future standard and core of the JavaScript language. Modern JavaScript continues to follow the standard while the standard itself has evolved together with the language over the years.

The effective standardisation allowed JavaScript to branch out into new developments around the now solid core language. Modern web development frameworks such as React rely on a form of JavaScript both for server and client side functionality. On top of this, JavaScript's popularity has also been pushed by modern web server solutions, such as the Node web-server. Node enables JavaScript to be used as a fully functional server-side language.

Through the language's evolution, modern JavaScript has, therefore, become a fully fledged multi-paradigm language. Some of its features have evolved to support native metaprogramming inside the JavaScript runtime. For instance, JavaScript supports a metaprogramming concept it refers to as Proxies [12]. Proxies are runtime callback handlers that hook into the native functionality of runtime JavaScript objects. Through proxies JavaScript can, e.g., intercept a function call and dynamically change its functionality based on the runtime environment and state of the caller object.

Similarly, a JavaScript object can employ self reflection at runtime through the native Reflect library [12]. Through reflection a language object can analyse and alter its execution dynamically based on its own runtime state. Therefore, JavaScript code can create dynamic behaviours based on the structure of the objects generated at runtime, effectively furthering the languages ability to be metaprogrammed. [4], [12]

2.3 PHP & JavaScript: A Meta-level relationship

PHP and JavaScript share a meta-level relationship. The two languages execute in separate environments, i.e., on the client and the server-side. PHP takes the role of the server-side language that sources, generates and serves the client-side JavaScript. [3], [4]

Regardless of the separate runtime environments, the two languages can still interact through the meta-level relationship provided by their shared lexical environment. A PHP source file can contain both PHP, JavaScript and other embedded languages, namely HTML and CSS. The JavaScript inside a PHP source file is separated into its own lexical scope, inside a `<script>` block. The PHP runtime is, however, able to interact with the contents of this script block during the server-side runtime. PHP is, therefore, given the ability to operate on the lexical

nature of JavaScript. This interaction can be considered as a meta-level relationship, where PHP is able to affect the underlying generation and future execution of the JavaScript code. [4], [11]

The meta-level relationship is further enhanced by the fact that PHP and JavaScript operate in different runtime environments. The server-side execution and evaluation of the PHP code provides a completely separate step for metaprogramming. During the evaluation of PHP code the client side JavaScript can be analysed and modified before the final code is staged for deployment. Therefore, through the meta-level relationship an organic multistage language and compilation are born [5]. The first evaluation happens during the interpretation of the PHP server-side code and the second phase is executed on the client-side JavaScript browser.

The generated JavaScript block is executed normally on the client-side browser. The execution is based on the standardised JavaScript syntax that PHP runtime has to conform to during the modifications of the script blocks. PHP, however, does not have a syntax-level understanding of the underlying JavaScript code, making modifications dangerous. [3], [4]

Regardless, the multistage nature and metaprogrammability of the two languages can be utilised in different ways. For instance, the separated runtime allows PHP to pass static variables to the underlying JavaScript before it is deployed to the client-side for execution. Similarly, conditional evaluation can be achieved through wrapping a block of JavaScript inside a PHP if-clause. In this way PHP can determine, on a lexical level, which JavaScript code is sent for evaluation on the client-side browser. [3], [4]

However, here stand the limits of this relationship. The lack of syntactical understanding makes variable exchange and other existing metaprogramming features unreliable at best. The metaprogramming itself is additionally only achieved through string operations provided by the PHP language. The hacky nature of these opera-

tions makes the utilisation of the organic interoperability buggy and often unusable. For instance PHP needs to utilise the echo and print statements in order to interact and modify the JavaScript during runtime. Such actions cause side-effects and are therefore commonly considered an anti-pattern in the PHP development community. Similarly, more uncommon features of the relationship, such as PHP's ability to define JavaScript variables and functions, often manifest themselves as a gimmicky side-effect of the server to client side relationship, limiting their effective usability.

This thesis, therefore, argues that with additional improvements upon the existing meta-level relationship one can grow the relationship towards an effective and safe metaprogramming interface. Only by building upon the organic metaprogrammability between the languages, one can create natural, yet powerful metaprogramming features between the two languages. With sufficient security and reliability through, e.g., custom syntax, the effectiveness of the relationship can also be maximised while the downsides are minimised. The actions toward this enhancement are further described in the following chapter.

3 Metaprogramming

Metaprogramming [6] can be defined as the development of software programs that are capable of accessing and treating other programs as data. Such programs are referred to as metaprograms. Metaprograms possess the ability to act as parent programs that are capable of analysing and operating upon the lexical nature of a child program. In more concrete terms, this ability enables a parent program to interact and operate upon a child program's source code.

The parent program is able to employ this ability in a plethora of different ways. The parent can, for instance, generate new program code directly into the source code of the child program. Furthermore, this generation can occur either by modifying the existing source code of the child program or by inserting completely new code into the child program's source code. Other meta-level methodologies available to the parent program include macro expansion, aspect oriented programming and meta-level traits. Metaprogramming methodologies are further discussed in chapter 3.1.

The parent program's ability to modify and interact with the child program results in a meta relationship between the parent and the child program. In this relationship the parent program is referred to as the metaprogram [1], as it possesses the ability to metaprogram the child. Often the child program is referred to as the subject program [1], as it is subject to the meta-level changes orchestrated by the parent of the meta-level relationship. With the above terms, metaprogram-

ming can be defined as the programmability of a meta-level relationship between a metaprogram and a subject program.

A metaprogram and the subject program it has control over can interact through their relationship on many different levels. The meta-level relationship, therefore, has the ability to alter and affect all stages of a metaprogram's life cycle. [1], [6], [10]

3.1 Metaprogramming methodology

Metaprogramming methodologies can be classified into different categories based on the different metaprogramming features and methods employed in the meta-level relationship between the meta and subject program. For instance, at the core of metaprogramming lies the creation of macros [1], [13], which were first popularised by their inclusion in the C-preprocessor and LISP language. Macros rely on the generative aspects of metaprogramming, while enabling the dynamic creation of additional source code. On the other hand, many high level modern languages like Python support reflection, which allows the subject program to inspect and act upon its own runtime structures.

The different metaprogramming methodology categories, therefore, target specialised problems within the field of software development. The problems can vary from, e.g., automatic code generation to the separation of concerns. Based on the goals of the metaprogramming system and the used methodologies, a metaprogram can be split into the following six categories: macro systems (macros), reflection, metaobject protocols, aspect-oriented programming, generative programming and multistage programming. [1], [6], [13]

3.1.1 Macro systems

Macro systems, or macros [13], are language level features that allow a developer to generate additional code inside a subject program. The underlying idea is to expand the subject code's functionality at run-time, by allowing the developer to compile meta-level structures onto the subject's source code. These meta-level structures, referred to as macros, are often embedded in the source code itself and employed through an external compiler such as a preprocessor.

The generation of subject code through meta-level macro structures is referred to as macro expansion [13]. In macro expansion the pre-defined macro code is expanded into the underlying subject program's source code. The expansion is done according to the employed macro language. During macro expansion the subject language is processed by a macro expander, which parses the high level macro structures into the subject language, acting effectively like a compiler or a transpiler. The process of macro expansion is iterative: the parsing is continued until all macro level invocations and structures are mapped into valid subject-level syntax. After the macro expansion is complete, the remaining subject language code is ready for compilation and execution by the subject language compiler.

Macro systems can be split into two categories based on how the macro expansion and macro level structures are employed. Lexical macros [13] are the first of the two categories. These kind of macro systems were originally introduced with the C-preprocessor. Lexical macros operate on a very basic lexical, i.e., textual level inside the subject code. Lexical macros provide macro expansion by examining the subject code as a sequence of tokens. The approach makes lexical macro expansion language agnostic but dependent on an external macro processor; the C-language and the C-preprocessor are a well known example. A macro processor executes macro expansion by examining the subject language for macro tokens and then transpiles them into syntactically correct subject language code. [13]

Syntactic macros [13], the second category of macro systems, are aware of the subject language's syntax. This makes them language specific structures often embedded into the subject language itself. For instance the Lisp language allows the developer to treat any piece of subject code the same as any internal data structure. This approach enables syntax aware macro expansion natively inside the subject language itself. Syntactic macros, therefore, remove the reliance on an external language processor, while requiring both subject and meta-level syntax knowledge from the subject language compiler. [13]

Additionally, macro systems can execute macro expansion either in a procedural, pattern based or hygienic manner [1]. All three define additional features any macro systems can employ depending on their implementation. In procedural macro expansion, a macro system can generate subject code based on internal computations of the language processor, which are based on macro invocations. Pattern based macro expansion is purely based on the replacement of macro syntax according to internal substitution mappings of the macro system. [1], [6]

Hygienic macro expansion is a solution to an inherent problem created by the nature of macro expansion. When a macro system expands the macro syntax, variables or other namespaces generated by the expansion can conflict with the underlying subject code. The problem is referred to as variable capture. In order to solve the variable capture problem, macro systems need to hygienically generate reserved subject language names during the macro expansion process. In hygienic macro expansion, a macro system effectively renames reserved names generated by the macro expansion. The renaming is done in a way that guarantees that it does not conflict with the underlying subject code. Scheme was one of the first languages with a hygienic macro system, after which hygienic macro expansion has become a stable feature of any generative metaprogramming solutions. [1], [6], [13]

3.1.2 Reflection

Reflection [1], [14] refers to a software system's ability to reflect upon its own structure and act upon that structure. Effectively this allows a reflective metaprogramming system to perform computations on its own behaviours during run-time. A reflective metaprogramming system is able to adjust its internal behaviour based on the dynamic requirements of its own execution. [1], [14]

The most basic form of reflection is the ability for introspective reflection [14]. A system that supports introspective reflection is able to read its internal meta-level structures during runtime. The ability to reflect upon the metaprogram's own self-representation allows the introspective system to change the course of subject code execution, based on the state of these structures. For instance, an introspective system can alter subject code execution depending on if a runtime object contains a certain required attribute. [1], [14]

On the other hand, reflective systems can also have the ability to alter their self-representation during runtime. This ability is referred to as intercession [14] and it describes a reflective system's ability to modify its internal structure during the subject code's execution. [1], [14]

Introspection and intercession are tightly tied together. A reflective system that can employ both introspective and intercessive reflection can both observe and modify its own meta-level structures during runtime. These meta-level structures, such as class attributes, remain otherwise inaccessible to regular subject code after initial compilation. However, a fully reflective system can, e.g., dynamically generate a class attribute if it determines it as missing. [1], [14]

A software system's ability to reflect upon itself can further be described based on the nature of the reflection itself. Structural reflection [14] allows the system to act introspectively against the structure of the subject code. Structure in this context represents the runtime structures defined by the subject code, e.g. classes,

methods and the internal attributes of these objects.

Software structures, on the other hand, create certain behaviours during their execution at runtime. Behavioural reflection [14] allows reflective systems to interact with and hook into these behaviours during runtime. Behaviorally reflective metaprogramming systems can, e.g., execute code during certain variable assignments, object creations or method invocations. [1], [6], [14]

3.1.3 Metaobject protocols

Metaprogramming can be embedded in the creation of a programming language. Such metaprogramming languages can expose powerful metaprogramming paradigms to the underlying subject language. Metaobject protocols (MOPs) [1], [10], [15] are an example of one such paradigm.

Metaobject protocols enable a metaprogram to interact with and alter the underlying object system of the subject language. In order to achieve this, MOPs need to employ meta-level APIs provided by the subject language itself. This approach enables the metaprogram to affect the underlying object system at the level of the subject language definition. Such access enables a metaprogram to implement features such as reflection, presented in section 3.1.2, but also gives it the ability to alter classes and object structures during runtime. [1], [15]

With meta object protocols a language can be considered to have meta-level class definitions that all subject language structures inherit from. The meta-level classes [10] are exposed to the metaprogram via the subject language's API. By modifying meta-classes and their behaviours, the metaprogram can modify all regular classes of the subject language. Therefore, the regular subject classes can be considered as instances of the meta-class, i.e., meta-objects. In this structure meta-classes are responsible for the behaviour and structure of the underlying meta-objects, allowing the metaprogram to populate the meta-objects with, e.g., new methods by modifying

the meta-class. [6], [10]

The above approach is indicative of meta-class based MOP systems. Some of the initial systems implementing the MOP methodology were Smalltalk and LISP, while modern examples contain, e.g., the Python MOP system [1]. Other approaches to MOPs include the metaobject-based MOP systems [1]. In the metaobject based approach the generated metaobjects share a common subject class for structural purposes, but their behaviours are defined by separate meta-level objects. [1], [6]

3.1.4 Aspect-oriented programming

Aspect-oriented programming (AOP) [1], [16] is a metaprogramming method that seeks to deal with the complexity of software development. Software development often deals with a large amount of concerns or features that increase the complexity of the subject program. This complexity needs to be effectively managed and limited where possible. AOP systems employ metaprogramming methodologies to limit the complexity of the subject program by targeting especially so called crosscutting concerns in the subject program. [1], [16]

Crosscutting concerns [16] refer to concerns that are common between different features of a subject program. AOPs split these concerns into separate and modular units referred to as aspects. The underlying idea is to insert these aspects into specific points in the subject program, called join points [16], at meta-level. The approach aims for allowing developers to express cross-cutting concerns in a modular way. [1]

Each aspect contains the program logic of their given concern and identifying information. This information referred to as advice, will be inserted into join points in the subject program based on predefined matching criteria. The process of connecting the meta-level aspects with their matching pointcuts is referred to as weaving. [16]

Weaving is a separate process specific to the metaprogramming of AOPs. Weaving is often accomplished with a separate compiler outside of the subject language. Such a separate compiler is referred to as an aspect weaver and it is used with static AOP, where the meta-language and subject language are separate. Dynamic AOP, on the other hand, allows for more freedom in terms of aspect weaving as the weaving occurs during runtime. Dynamic AOP has a larger overhead and is less efficient than static AOP. [1], [16]

Seen from the lens of the subject language, AOPs effectively insert code defined on a meta-level to certain points in the program's execution. Each aspect can be inserted either around, before or after a pointcut matching the criteria of the weaved aspect. The code itself can contain anything from method and variable declarations to code meant to be executed, e.g., before a certain function call. This feature makes AOP an effective measure of delivering metaprogramming solutions to subject code. However, AOP weaving often already relies on existing metaprogramming methods, highlighting the convoluted nature of the topic. [6], [16]

Aspect oriented programming can be considered a form of meta-level reflection, as discussed in 3.1.2. While reflection is an overarching solution targeted at meta-level problems, AOP is an example of specialised metaprogramming. AOP enables the management of crosscutting concerns with straightforward and safe results, while knowingly sacrificing the expressiveness of other algorithmic methods. [1]

3.1.5 Generative programming

Generative programming [1], [16] refers to a programming paradigm that seeks to enable automatic manufacturing of an end-product's subject code. To achieve such a goal the paradigm requires software systems to be organised in elementary and reusable components. Organising the software in this manner enables the generation of arbitrary yet often highly specific subject code with its own requirements of

optimization and domain specificity. [1], [16]

Template systems [17], such as the C++ templates, are a common representative of generative metaprogramming. Template systems enable metaprogramming through the creation of template structures, e.g., the skeletal structures of classes and functions. Templates are then fulfilled through concrete parameters allowing for subject code generations and generalisation. Templates create a strict, yet safe, environment for the generation of source code, since they enforce subject code generation through well structured templates and in turn disallow the generation of other free form subject code. [17]

AST transformations, on the other hand, allow for the generation of arbitrary subject code. In AST transformations subject code is generated through injected quasi-quotations that are evaluated into effective subject code [1], [6]. Subsequently, systems that enable AST transformations also enable the meta-level system to traverse the AST in order to enable further metaprogramming structures. Such AST transformations can also present themselves in the form of code annotations. Code annotations modify the compiler's interpretation of the subject code, allowing for the generation of compiler level changes to the evaluated source code. [1], [6]

In a similar manner some generative metaprogramming systems enable compile-time reflection in order to achieve subject code generation [9], [16]. The main incentive of compile-time reflection is the reliance on existing code structures in order to achieve static type safety inside the generated source code. For instance, during compile-time reflection the metaprogramming system can alter the way generic classes are compiled based on certain predicates, resulting in the conditional generation of differing subject code. Similarly some systems allow for the traversal of the AST enabling compile time analysis of meta-level structures and therefore conditional generation of subject code. [9], [16]

A more common approach to compiler level code generation is the employment

of Class compositions [1]. Class compositions refer to meta-level structures that enhance existing subject code classes through external code injection. Traits and mix-ins are, therefore, a common form of generative metaprogramming. Traits are type safe extensions to existing class structures that enable the insertion of arbitrary functionality, e.g., in the form of additional class functions and variables. Traits are evaluated during a class's compilation, enabling further functionality without affecting the type safety of the subject class. Traits, therefore, enable compiler level code generation without the need to further access or traverse the AST. The logic of traits can be further extended to feature-oriented programming, where traits, i.e., features are able to extend and create new classes inside subject code while retaining type safety. [17], [18]

3.1.6 Multistage programming

Multistage programming [5], [7], [9] introduces the concept of program evaluation levels. The underlying idea is to split a source program into different stages, each of which are meant to deliver a version of the compiled subject program. Each stage is evaluated separately based on the pre-defined staging annotations inside each of the evaluation stages. Staging annotations [5] refer to the meta-level structures that specify the order of evaluation between different program stages. The annotations effectively enable the creation of delayed, future stages within a subject program. [5], [9], [19]

Multistage programming enables metaprogramming through the means of partial evaluation and program specialisation. Through the different stages of the source program, a generic program can be specialised into a highly specific program using parameterization [17]. In parameterization the source program is organised in a generic fashion allowing for the explicit specialisation in future stages of the multi-stage compilation process. The specialisation is, in turn, executed throughout the

compilation of the separate program stages. On top of enabling specialisation this approach also removes load from the runtime stages by moving program specialisation and functionality into the compilation of the metaprogram. [1], [17]

Partial evaluation [8] is another metaprogramming approach specific to multistage programs. In partial evaluation the source program contains specific areas, such as quasi-quotes, that define the delayed stages inside the source program. When the designated partial evaluation stage occurs, newly generated subject code can be injected into the generated source stage. This evaluation can happen in the current stage of the multistage cycle, making partial evaluation a tool similar to macro expansion. The separation between the two methods remains in the technique of definition. Whereas macro systems use meta-level syntax in order to define the syntax of the macro expansion, the multistage programming approach uses it to define the borders of the delayed stage, in which the compilation of the generated multistage source code occurs. [1], [8]

Furthermore, multistage programming implementations can differ in their approach to the count and definition of the separate evaluation stages [1]. Many multistage approaches that implement the meta-level structures necessary to support multi stage programming also support an infinite number of evaluation stages. Some of these approaches limit the number of stages, for instance, to exactly two. Enforcing limitations effectively limits the complexity of the multistage program, but with the added cost of also limiting the employment of other meta-level benefits. [1], [9], [19]

4 Multistage languages

Multistage languages [5], [8], [9] are ones that support the multistage programming paradigm discussed in chapter 3.1.6. As a metaprogramming paradigm, multistage programming provides an avenue for the meta-level manipulation of a language's source code. In a multistage language based approach, the subject language consists of one or multiple individual language stages. Therefore, a multistage language structure occurs when a parent program has the ability to create delayed child stages and affect their compilation. In the context of multistage languages the parent program, i.e., the metaprogram can be referred to as the metastage. Similarly, instead of a singular subject program, a multistage language can have multiple subject stages inside the meta-level relationship. Multistage programming can, therefore, be further defined as the organisation of a metaprogram into multiples of meta- and subject stages that all share a meta-level relationship. [7], [9], [19]

4.1 Multistage Methodology

The creation of a multistage language begins with the separation of stages. In stage separation, i.e., staging, a source program is split into multiple stages according to the principles of metaprogramming and multistage programming. The goal of program staging is to reduce the cost of runtime load that is often associated with other methods of metaprogramming and dynamic code generation. [5], [8]

In order to implement staging, a multistage language requires a functioning

methodology for stage generation, evaluation and execution. Many modern languages support some features of metaprogramming but lack the core features required for the generation of a multistage language. For instance, in order to create delayed subject stages a multistage language requires the ability to create a zone of delayed execution inside the initial meta-level stage. Through delayed execution multistage programming can then take hold and the language naturally constructs itself into meta- and subject stages. Similarly, the delayed subject code requires a method of dynamic evaluation and thereafter execution in order to become functional. In order to implement the above functionalities, the language needs to first support staging through native structures or implement them through a separate multistage language generator. [5], [8], [9]

MetaOCaml [5], [8] is an early adopter of the multistage language approach. The language implements the required features of a multistage language mainly through native language syntax. In MetaOCaml a delayed execution can be constructed with the Bracket syntax `(.< subject-code >.)`. The brackets act as staging annotation, marking the syntax inside them for later execution. [5], [8]

```
# Normal assignment -> int = 7
let a = 3+4;;
# Delayed assignment -> code = .<3+4>.
let a = .<3+4>.;;
```

Listing 4.1: MetaOCaml Quasi-quotations.

In the above code example 4.1 a variable definition is executed first in a regular arithmetic fashion. Then the same operation is executed using the bracket syntax. The commented results demonstrate how the simple arithmetic operation is executed immediately while the bracket definition is stored as a code entity for later execution. The same operation inside the bracket syntax is, therefore, delayed until a future operation compiles the result of the delayed clause. Hence, this approach can be used to create the required subject stages of the multistage language.

In MetaOCaml the execution and evaluation of delayed operations occur through the `run` syntax. The `run` syntax enables a stored or dynamically generated statement to be evaluated and executed at an arbitrary stage of the multistage program. In this context the execution can occur on any of the subject stages generated after the initial meta-stage. [5], [8]

The metaprogramming capability of a multistage language can be further improved by enabling nesting and just-in-time (JIT) compilation of the delayed statements. In the context of multistage languages, nesting refers to the meta-stage's ability to store uncompiled code in subsequent layers. Just in time compilation, on the other hand, refers to the program's ability to compile a piece of subject code dynamically when evoked upon. [5], [9]

In MetaOCaml these features are provided through the usage of an escape statement `(. staged-code)`. The escape statement effectively allows for the nesting of existing delay fragments into a combined expression. When the expression is evaluated it JIT compiles any references to included staged fragments, creating the final statement. [5], [8]

```
# Delayed assignment
# int code = .<3+4>.
let a = .<3+4>.;;

# Nested assignment:
# code = .<(3+4) * (3+4)>.
let b = .<.~a * .~a >. ;;

# Executed operation
# int = 9
let c = .! b;;
```

Listing 4.2: MetaOCaml language features.

In the above example 4.2 the variable `a` is first initialised with a delayed statement, containing an arithmetic operation. Statement `b` is thereafter initialised with another delayed operation, however, the delayed stage now contains references to the delayed operation `a`. As demonstrated by the comments, the nested statements are

evaluated into the context of statement `b`, which remains as a delayed statement. The value of `b` can thereafter be resolved using the `run` operator, as seen in the assignment to the variable `c`. Once the delayed statement is evaluated through the `run` operator the delayed operation is first compiled and then evaluated to produce the final value.

In MetaOCaml multistage programming manifests in a pure form. It is, however, not actively used. Terra, is an example of a modern implementation of a multistage language. The language is designed as a system level language capable of efficient low-level operations, e.g., memory management. Terra can be run as a standalone language or, more notably, metaprogrammed by the Lua scripting language. The two languages together create a multistage language, yet separately they don't employ the mechanics of multistage programming. Therefore, the multistage language and subsequent meta-level relationship is built using existing functionalities on top of the relationship the two languages share. [7], [19]

Terra [20] implements staging using a predefined meta-level keyword, `terra`. The Lua compiler recognizes this keyword and delays the execution of the Terra stage. The Terra stage will be later executed by the Terra runtime that also compiles the code. The staging annotation is created through the foreign language interface of Lua, enabling the relationship to be extended to a meta-level relationship. The example below 4.3, sourced from [7], demonstrates how a Terra stage is defined inside Lua code.

```
terra  min(a: int, b: int) : int
    if  a < b  then return  a
    else return  b  end
end
```

Listing 4.3: Terra code scoping.

Terra can also be used to demonstrate other multistage features present in MetaOCaml. For instance, Terra enables functions defined inside the Terra stage to

be natively called from Lua. The invocation happens similarly to Lua function calls, except that the mix of Terra and Lua code is actively JIT compiled for the execution of the function. Therefore, any meta-level structures present in Lua code are actively compiled during the function call providing a similar result to the MetaOCaml `run` function. Lua is also natively able to store Terra functions and stages inside Lua variables achieving the combination and metaprogrammability that the MetaOCaml `escape` statement provides. [7], [19], [21]

```
function ImageTemplate(PixelType)
    struct  ImageImplementation {
        data : &PixelType,
        N : int
    }

    terra  ImageImplementation:init(N: int): {}
        self.data = [&PixelType](std.malloc(N*N*sizeof(
           PixelType)))
        self.N = N
    end

    terra  ImageImplementation:get(x: int, y: int) : PixelType
        return  self.data[x*self.N + y]
    end

    return ImageImplementation
end
```

Listing 4.4: Terra & Lua Example

The above example 4.4, reproduced from [7], presents how a Lua function can be used to metaprogram a Terra implementation. Here the Lua function stores a collection of Terra stages defined with the Terra syntax. The purpose of the function is to dynamically create the required Terra structure type during the Lua runtime. This, in turn, demonstrates how Lua is able to dynamically metaprogram the structure of Terra code based on the Lua runtime `PixelType` input. [7], [19], [20]

4.2 Implementation types

Regardless of their core similarities, Terra and MetaOCaml represent different implementations of the multistage language paradigm. Multistage languages can be differentiated, for instance, based on their lexical environments and the implementation there of [1]. In the lexical context Terra takes advantage of the relationship between two different languages where the meta-language, i.e., Lua, remains a separate entity from the subject language of Terra. As per definition presented in [1], using separate languages inside the different stages of a multistage language makes the approach heterogeneous. Therefore, Terra represents a heterogeneous multistage language, where the meta- and subject stages are implemented with completely separate languages. On the other hand, MetaOCaml implements staging of both the meta- and subject stages inside the same language of MetaOCaml. Therefore MetaOCaml can be referred to as an homogeneous multistage language. [1], [5], [7]

Similarly multistage languages can take different approaches to how the meta and subject stages interact. For instance, both in Terra and MetaOCaml the subject stage is contained in the same lexical environment, while the delayed execution is produced through specially derived staging annotations. Therefore, both of the example languages implement a homogeneous [1] approach to how the meta and subject stages interact. [1], [5], [7]

Another possible approach to establishing interactions between the meta and subject stage is the heterogeneous approach. The heterogeneous approach to the interaction inside a multistage environment stores the meta-level and subsequent subject-level stages in completely separate source files and even types thereof. The heterogeneous approach is often a result of the natural relationship between languages and is therefore common in compilers and other heterogeneous multi stage programs. The heterogeneous approach increases clarity, but can in turn increase the complexity in the subject program's logic. [1], [5], [9]

Finally another noticeable difference between some multistage language implementations is their approach to the implementation of staging. MetaOCaml, for instance, provides a structure to create staging inside the core language itself. Through the methodology employed by MetaOCaml the number of stages can technically be unlimited. This means the meta stage can be followed by practically an unlimited amount of subject stages. On the other hand, Terra remains an example of a well defined approach. In Terra, staging is a part of the meta-level relationship shared between the Terra and Lua languages. Therefore, the numbers of stages remains limited to two, first being the Lua based meta stage and the latter being the Terra subject stage. [1], [7], [19]

4.3 Problems & Solutions

Multistage languages share a number of common issues due to their complex nature. These issues consist of, e.g., managing the interactions between the different stages and the combination of different language structures, which become especially crucial to be managed when a multistage program is developed. [5], [8]

At the core of every multistage language is the generation of new subject code. Code generation happens when a meta stage inserts generated source code dynamically into a subsequent subject stage. Code generation, however, is only successful if the generated code can be executed by the subject runtime. This becomes especially troublesome with heterogeneous multistage programs as the meta stage is written in a separate language when compared to the subject stage. Therefore, issues can arise when the meta stage generates invalid subject code, effectively resulting in an error in its execution. [5], [8], [9]

A common strategy in multistage code generation is to treat the subject code on face value, i.e., as a lexical string representation. This approach can be effective when the amount of generated code is small. However, scaling up a purely lexical approach

quickly leads to issues with syntactical correctness, as especially in heterogeneous environments the meta-stage has no way of validating the generated code. The issue can be mitigated with strong typing that will ensure type-safety over the meta- to subject language interface. Similarly different code injection methods, such as quasi quotations, can be used to inject code more accurately into the subject stage. When type safe code is injected locally into a well defined part of the subject stage, the number of syntactical errors can be minimised. [5], [8]

The only way to fully mitigate the issues with code generation, however, consists of giving the meta-stage the ability to validate the generated subject code. In order to do such a validation, the meta stage requires an abstract syntax tree implementation of the subject language. An AST implementation basically evolves into a fully functional foreign language interface, which in turn can evaluate and validate the generated code before it is injected into the subject stage. [2], [5], [8]

Another issue relevant to multistage languages is the ability to transfer data between the meta and subject stages. Once more, in homogeneous multistage languages the transfer of data tends to be more reliable. For instance, homogeneous languages can more easily pass references or by default syntactically correct variable definitions to subsequent subject stages. On the other hand, heterogeneous languages, especially those consisting of separate run times, often lack the ability to directly pass references between the stages. As a result the meta-stage needs to ensure that both the transformation of existing data formats, e.g., custom classes are translated properly into the subject stages. This requirement is especially important on strongly typed subject languages where custom object transfers can become impossible without further connection between the two languages. [7], [8], [19]

A simple way of passing data between the meta- and subject stages is code generation. Through code generation simple variable statements can be passed by inserting a syntactically correct subject language statement into the underlying

stage. Raw memory references on the other hand become possible when both of the languages function on a low enough level to access and pass direct memory references between the stages. [7], [8], [19]

While injecting new variables into subject stages, multistage languages also need to deal with variable capture, similarly to macro based systems discussed in 3.1.1. Variable capture occurs when an injected variable shares the same name with a pre-existing subject stage variable. A name collision occurs, which, depending on the subject language, can either result in an error or other unintended side-effects. Therefore, multistage languages need to deal with the concept of hygiene, familiar from the macro based approaches. For instance, variables can be hygienically injected by dynamically renaming the injected variable. The process of renaming, however, requires further management on the side of the meta stage, which now is responsible for renaming each reference to the given variable. [5], [8], [9]

All of the above approaches culminate in one of the requirements of modern development, i.e., error handling and parsing. Multistage languages struggle with presenting errors in a concise manner, as the error can occur multiple stages deep into the language stack. In an ideal situation errors could only occur on the meta-stage, as debugging such errors remain similar to debugging a regular language. This ideal, however, requires each meta stage to check their immediate subject stages for errors in a recursive process. The approach is complex, but would result in a relatively maintainable multistage language. The added complexity, however, becomes harder to maintain when approaches of unlimited stages and heterogeneous languages are considered. Such approaches require special handling for how error stacks are generated and in turn presented in a maintainable and clear manner. [1], [5], [8]

5 PScript: Introduction

As discussed in chapter 2, PHP [3] and JavaScript [4] share a meta-level relationship. The relationship between the languages ultimately manifests as exploitative side-effects that allow JavaScript to be manipulated by PHP. The purpose of this thesis is to improve upon the existing relationship between the two languages by transforming the relationship's features from perhaps unintended side-effects to secure features. To achieve this goal, a metaprogramming solution, referred to as PScript, is developed.

PScript is built to tie the loose ends of the PHP-JavaScript relationship into a new, more complete, multi-language solution. The multi-language solution is built through a non-intrusive language processor functioning on top of the PHP runtime. The PScript language processor implements the preprocessor [2] pattern and, therefore, acts as an additional processing step before the PHP interpreter.

The PScript runtime is written as a PHP library using the native PHP version 8.2 [11]. The PScript runtime can therefore be enabled in any existing PHP project using the native language imports. However, due to its reliance on PHP host, PScript only creates a pseudo runtime inside the PHP host instead of a completely separate preprocessor.

The preprocessing approach is especially apparent while including the PScript runtime into an existing PHP project. For instance, each dynamic import inside the PHP host can be ran through the PScript preprocessor on an individual ba-

sis. The end result of processed files are native PHP source files that can be executed natively inside the PHP runtime. For clarity, PScript introduces a separate non-mandatory file format for unprocessed PScript source code. The files are marked with a `.pscript` extension and can contain proprietary PScript syntax, PHP, JavaScript and HTML. In addition, the PScript runtime can also be embedded into any existing PHP file in order to manipulate existing PHP source code, as long as the file is then imported through the preprocessor. [2]

As a preprocessor solution, PScript offers a defined set of syntax designed to manipulate and extend the PHP-JavaScript relationship. Internally the language relies on the PHP and JavaScript runtimes for error handling, interpretation and execution of transpiled source code. This approach allows PScript to focus on providing a pre-designed set of improvements in the form of meta-level extensions and novel syntax definitions. [2], [7]

PScript syntax and features are oriented towards improving the PHP-JavaScript relationship. These improvements are primarily done through providing the existing language environment with new syntax. The PScript runtime specifically provides syntax for hygienically injecting PHP variables into JavaScript code, improving the multi-language [5] environment through clearer scoping and for injecting PHP expressions into JavaScript code through quasi-quotations [22]. Each of the above features share the combined goal of bridging the client-to-server interface between PHP and JavaScript, while simultaneously improving upon it. [7], [10], [20]

PScript's other major goal is to extend the metaprogrammability of the PHP-JavaScript relationship. Metaprogrammability, in this context, refers to the number of metaprogramming features present in the multi-language relationship. By default the PHP-JavaScript relationship supports metaprogramming as a side-effect of the server-to-client interface discussed above. PScript builds upon this interface by introducing new syntax capable of proving additional metaprogramming

features. These features include, e.g., multistage-programming [5] features such as conditional compilation [9] and meta-level namespaces [7], [20], as well as macro based approaches [13] such as specialized templates [17] and traits [18].

Example 5.1 demonstrates a traditional hello-world example using PScript. The example represents a complete PScript file, where the `client` block contains PScript enhanced JavaScript and the `pscript` block contains PScript enhanced PHP.

A complete list of PScript language features, syntax and their implementations are introduced in chapter 7. Additionally, a real-world example using a PScript based solution is demonstrated in chapter 8.

```
<?pscript
    $print_clause = true;
    $print_text = "Hello world!";
?>

client {
    const print_text = $print_text;

    if ($print_clause) {
        console.log(print_text);
    }
}
```

Listing 5.1: PScript: Hello world with conditional compilation.

6 PScript: Design goals

As described in chapter 5, PScript's purpose is to build upon the relationship between the two languages of PHP [3] and JavaScript [4]. In order to achieve this goal, PScript was designed as a functional language extension to PHP. Therefore, the design goals discussed in this chapter emphasise the ease of use and reliability inside any existing PHP applications. The overall goal of the PScript extension remains the utilisation and improvement of the existing meta-level relationship. Therefore, the design goals also reflect the need for both optimization of existing features and for the creation of new meta-level improvements. [1]

6.1 Clarity

PScript at its core is a multistage language [5], [8]. The multistage host consists of the meta-stage PHP code and the subject stage JavaScript payload. Therefore, the first design goal seeks to address the clarity of this relationship, when the added complexity of the PScript runtime is introduced.

PHP and JavaScript are clearly separate languages, with completely separate runtimes. Introducing an additional language and runtime into this ordeal could significantly affect the clarity of the end product. Therefore, the decision of treating PScript purely as an extension of PHP was made. Treating PScript as a language extension both reduces the complexity of the solution while improving its clarity.

Regardless of the aforementioned precaution the line between the two languages

becomes increasingly blurred when PScript is introduced. In order to ensure clarity inside a PHP source file, PScript syntax should remain clearly separated from both of the existing run-times. Therefore, the preprocessor [2] pattern was chosen, as it allows PScript files to be evaluated during the runtime of the server side PHP code. Preprocessor in this context refers to PScript's ability to execute inside the PHP runtime process, translating the embedded PScript syntax into processed PHP files.

The clarity of a native language solution can only be achieved if PScript also follows existing PHP conventions. When implemented as a language extension, PScript can remain as an active part of the PHP language by allowing, e.g., the creation of new syntax and other novel language features. New features, in turn, remain key for improving the reliability of the relationship between PHP and JavaScript. However, creating a completely new syntax tree [2] inside the already crowded source file can create additional issues for clarity. In order to avoid such a result, PScript syntax should remain limited and clearly separated from any existing source syntax. The separation enforces the lines between each step of the multistage program improving the clarity thereof [7]. At the same time, any newly created syntax definition should coincide with that of native PHP in order to remain a natural part of PHP development. This remains a requirement of clarity, while PScript is attached to the PHP runtime and targets existing PHP based projects. [7], [20]

One can also consider the clarity of an interpreted PScript file. For instance when PScript interacts with the PHP source on a meta-level, the produced PHP source files can remain very obscure. Therefore, when the meta-level structures [1] of PScript are expanded upon, the solution should ensure that the resulting PHP and JavaScript code remain readable. Additionally, similar language processors often minimise and lexically optimise the produced source code. In the case of PScript, such approaches would significantly affect clarity as the processed PHP source code would remain unreadable in the case it requires further debugging. [2], [8]

```
<?pscript
    $print_clause = true;
    $print_text = "Hello world!";
?>

client {
    const print_clause = $print_clause;
    const print_text = $print_text;

    if (print_clause) {
        console.log(print_text);
    }
}
```

Listing 6.1: PScript: Clarity of runtime separation.

The example 6.1 demonstrates the separation PScript creates between each language runtime. PScript enhanced PHP remains inside a `<?pscript ?>` code block similar to the native PHP alternative, while the `client` keyword is used to define a PScript runtime block. The PScript `client` block scoping follows the PHP standard by using the `{ }` bracket syntax, while the `client` keyword is used to remind the user that the code inside the scope is meant specifically for client-side usage.

Example 6.2 shows the transpiled end-result of the original source code. Clarity is maintained by not minimizing the resulting source code and maintaining the original order of execution. Similarly, the transferred variables defined as `print_clause` and `print_text`, are transpiled hygienically [13] into JavaScript while maintaining the original naming of each variable. Each code block, in turn, is transformed into native PHP-HTML format, but subsequently marked with unique PScript tags, which refer to the order of transpilation. E.g. the first PScript block in the example is transpiled into a HTML `<script>` block and tagged with the PScript identifier `id="pscript-block-0"` .

```
<?php
    $print_clause = true;
    $print_text = "Hello world!";
?>

<script id="pscript-block-0">
    const print_text_K7429691 = "Hello world!";
    const print_clause_a667n925 = true;
```

```
    const print_clause = print_clause_a667n925;
    const print_text = print_text_K7429691;
    if (print_clause) {
        console.log(print_text);
    }
</script>
```

Listing 6.2: PScript: Clarity of transpiled source code.

6.2 Usability

In order to act as an effective extension of PHP, the PScript preprocessor [2] needs to feel familiar to any existing PHP developer. Usability remains especially important when designing the format of new syntax definitions. Each definition should be defined in a way that remains true to the lexical nature of PHP [11]. For instance, the usage of `{}` parentheses in block scoping remains a staple of the PHP syntax. Therefore, PScript syntax should also follow the this pattern among other PHP syntax sugar.

Additionally, in order to maximise usability, any new syntax should also remain minimal. By limiting the amount of unknown and unfamiliar syntax one can also minimise the required on-boarding and knowledge needed for the usage of PScript. At the same time, any new syntax should not be afraid to tackle the existing usability issues of PHP. E.g., the definition of JavaScript stages inside HTML script blocks remains a limiting factor for usability and clarity of existing PHP solution. [2]

Any file written in pure PHP [3], [11] should remain usable through the PScript preprocessor. Backwards compatibility ensures usability inside existing PHP projects, as developers can freely mix PScript files inside their existing PHP projects. On the other hand, PScript code needs to also function inside the PHP lexical environment. I.e., PScript code needs to remain easily importable into any existing PHP file and, vice versa, existing PHP code needs to function inside PScript files.

PScript should also not differ unnecessarily from the PHP host. For instance, creating PScript with a completely separate language would significantly reduce the usability of the project. Requiring developers to learn a third language in order to gain the benefits of the solution would create a block for usability. Another benefit arises from the maintenance of the language. When PScript is written in native PHP, any passive up-keep the language receives is most of the time extrapolated to the project itself.

```
<?pscript
    // Client block namespace definitions
    $namespace1 = [];

    // Client variable reference
    $namespace1['hello_world'] = client hello1;
?>

// Client block scoping
client {
    // Quasi quotation syntax
    $[create_animal("dog")]
}
```

Listing 6.3: PScript syntax listing.

The above example 6.3 demonstrates the limited amount of new syntax introduced by PScript. From the example one can observe how PScript first defines a meta-level namespace [20] and then inserts a cross language variable reference [7] using a mix of native PHP syntax and the PScript `client` keyword. Similarly, the example presents the client block scoping syntax `client {}` present in all PScript files, together with an additional example of the PScript quasi-quotation syntax `$[]` [22]. The amount of new syntax is limited to the above three, allowing PScript to be quickly introduced into an existing project.

Example 6.4 demonstrates PScript's ability to import existing PHP implementations directly into a PScript file, maximising usability. In example 6.4 the file `user.php` is imported into the PScript context allowing the existing PHP function `get_current_active_user` to be used directly.

```
<?pscript
    require_once( ROOT_PATH . '/public/demo/user.php');
    $user = get_current_active_user();
?>
```

Listing 6.4: PScript: Usage of existing PHP code.

Example 6.5 demonstrates how PScript files are imported into the existing PHP host. The PScript preprocessor runtime is first imported with the `require` statement. This allows the runtime to be used at any point in the following PHP execution. Using native PHP syntax sugar, the PScript preprocessor can also be instantiated into the active namespace with the `use` statement. In order to maximize usability, the PScript runtime can be chained with the native `require` statement in a natural manner. In example 6.5 the `demo.pscript` file is imported using the PScript runtime by chaining the two require statements together. As stated by the code, first the file is imported and ran through the PScript runtime and thereafter into the subsequent PHP runtime.

```
<?php
require_once(ROOT_PATH . "/processor/PScript.php");

use \PScript\PScript;

$pscript_file_path = "/public/demo/demo.pscript";

require_once(PScript::require($pscript_file_path));
?>
```

Listing 6.5: PScript: Preprocessor runtime initialisation.

6.3 Efficiency

PScript should remain an efficient extension to the existing PHP host [3]. Efficiency becomes a natural consideration when PScript extends already optimised language processors with additional features and processing. These efficiency concerns can be mitigated by, e.g., not requiring a separate compilation step for the usage of PScript.

Instead, the preprocessor is made to run dynamically during the PHP runtime, allowing PScript to benefit from any optimizations of the existing PHP interpreter. The dynamic approach also optimises the preprocessor when it is integrated as a part of an existing PHP based workflow.

The computational load produced by PScript's pseudo runtime [2] remains in the interpretation and transpilation of PScript syntax. Especially the continuous interpretation of PScript files becomes redundant when the produced PHP files rarely change in actual deployment. In order to mitigate this issue, sufficient caching should be implemented. By employing efficient caching of the translated PScript source files, one can significantly reduce the amount of runtime load required to parse each PScript file. Therefore, each PScript file is interpreted only once whenever changes appear inside the original source code. This creates an active cache system that is aware of the state of the original PHP source code and their changes.

Another consideration of efficiency remains in the end result of PScript's transpilation process [2]. The PHP source code produced by PScript should remain in an optimised format. As discussed above in chapter 6.1, the optimization of the generated source should not occur at the cost of readability. Instead the optimised source can, for instance, remove duplicate variable definitions, unnecessary references and other language constructs that are not needed in the subject program. [2], [3], [11]

The table 6.1 shows a set of measurements of the PScript runtime. Each measurement was taken through a dockerized PHP-PScript container using the PHP version 8.2 and measured in microseconds. The measured code and its functionalities are further presented in 8.

From the measurements one can see that, on average, PScript has an effect of 0,00066 seconds on the PHP runtime. When compared to the average of cached PScript load, which effectively represents a native PHP load time, the increase is around 37%. Therefore, while dynamically generating the PHP source on every run-

Table 6.1: PScript: Runtime transpilation efficiency measurements.

	Dynamic transpilation	Cached transpilation	Difference
	0.002378	0.001794	0.000584
	0.002537	0.001938	0.000599
	0.002403	0.001791	0.000612
	0.002186	0.001546	0.000640
	0.002946	0.002115	0.000831
	0.001882	0.001652	0.000230
	0.002520	0.001471	0.001049
	0.003013	0.002166	0.000847
	0.001828	0.001630	0.000197
	0.002602	0.001558	0.001044
Averages	0.002430	0.001767	0.000664

time, the runtime load remains around third of the original load time. Furthermore, under normal usage, the increased runtime load occurs only once, after which the cached load times take effect.

6.4 Security

Security is key in any modern web-application. The application server exists as a secure zone, protected often by extensive security features. One of the weak points in this zone remains the required interaction between the secure server and the, by nature, insecure client application. The existing relationship between JavaScript and PHP remains at the edge of this security zone. As PScript is able to manipulate both sides of the security zone at a meta-level, security should be a major goal of the solution.

For instance, server side code is home to many secrets in the form of environment variables and in some cases even regular variables. When PScript interacts with the PHP source code, it needs to make sure that variables that are not meant for client access are not transferred to JavaScript. Such features become especially important when PScript interacts with authentication or API security features, as such features

often trade information between the client and the server.

PScript, therefore, follows the general best practices in terms of application security. The reliance on server-side PHP provides PScript with added security features from the continuous up-keep of the host language. Regardless, the solution remains conscious of existing security weaknesses in order to not replicate previous mistakes or generate new attack vectors.

6.5 Improvements and fixes to smaller issues

The PHP-to-JavaScript relationship contains features that are usable in practice but are unreliable. As PScript seeks to improve upon the pre-existing relationship between the two languages, the language processor should try to address such problematic issues. The improvements on existing features should strive towards the same design goals as presented above, ensuring consistency of the solution.

As described in chapter 5, PHP and JavaScript occupy the same lexical environment. However, their relationship currently relies on the usage of another language, HTML. HTML provides the `<script>` blocks that all JavaScript code occupies. This dependency is demonstrated in a simple `hello_world` example in example 6.6

```
<?php
    $hello = "Hello world!";
    echo $hello;
?>

<script id="hello-world-script">
    const hello = "Hello world!"
    console.log(hello);
</script>
```

Listing 6.6: Native PHP script block usage.

PScript allows existing HTML dependencies to be replaced with native PScript syntax. PScript's block scoping is securely generated into the HTML variant with properly addressed variable definitions and other PScript features. At the same time

PScript provides the developer with a more straightforward development experience by removing the need to mix in HTML syntax. The PScript equivalent to example 6.6, demonstrated in 6.7, shows how the PScript runtime and subsequent block scoping improves upon the HTML equivalent, in a manner that better matches the syntax of the PHP host.

```
<?pscript
    $hello = "Hello world!";
    echo $hello;
?>

client {
    console.log($hello);
}
```

Listing 6.7: PScript enhanced client block scoping and variable transfer.

Another existing feature which requires improvement is the variable transfer between the two languages. Currently it remains as a side-effect of the relationship as PHP can employ the `echo` clause to print stringified values into JavaScript code. Technically this functionality can be used to export PHP values into JavaScript variables. However, the relationship doesn't allow for the usage of direct references or the evaluations thereof.

The improvements PScript provides regarding cross language variable transfer [7], [19], can be seen by comparing examples 6.7 and 6.8. In example 6.8 native PHP is used to `echo` a string value directly into JavaScript. Example 6.7 shows the same example using the PScript variable transfer mechanism. The example demonstrates how variable transfer can be securely supported and even improved by the PScript runtime generation process. Further examples regarding variable transfer mechanisms can be seen in chapter 7.2.

```
<?php
    $hello = "Hello world!";
    echo $hello;
?>

<script id="hello-world-script">
```

```
    console.log(
        <?php echo $hello; ?>
    );
</script>
```

Listing 6.8: Native PHP string variable transfer.

Similarly the PHP's role as the server side language can also be used to conditionally deliver JavaScript code blocks. The usage of this mechanism, however, remains clunky and the feature can't be considered true conditional compilation. Example 6.9 demonstrates native PHP based conditional delivery of a JavaScript code block, using the `echo` statement. PScript improves upon the above approach by removing the requirement for using PHP and HTML script blocks. Simultaneously PScript implements true conditional compilation of JavaScript code through the aforementioned PScript variable injection mechanism. Example 6.10 demonstrates the improvements PScript provides in comparison to example 6.9. [5], [7], [19]

```
<?php
    $hello = "Hello world!";
    $say_hello = false;
?>

<script id="hello-world-script">
    <?php if ($say_hello) {
        echo "
            const hello = '" . $hello . "';
            console.log(hello);
        ";
        }
    ?>
</script>
```

Listing 6.9: Native PHP conditional code delivery.

```
<?php
    $hello = "Hello world!";
    $say_hello = false;
?>

client {
    if ($say_hello) {
        console.log($hello);
    }
}
```

Listing 6.10: PScript conditional compilation.

As seen with the above sections, many of the native multistage operations inside the PHP-JavaScript relationships rely on PHP's ability to `echo` string variables into JavaScript code. PScript improves upon this approach by securing the generative operations [22] inside designated syntax definitions and a multistage language processor [7], [8]. Therefore, instead of relying on the primitive string injection features available to PHP, PScript employs language processing algorithms [2] similar to how compilers operate upon existing source code. As a result, PScript can support a wider range of generative features and metaprogramming methodologies. Examples and operations of the PScript language processor are further examined in chapter 7.1.

6.6 Metaprogrammability

Metaprogrammability [1], [6] remains at the core of the PHP-JavaScript relationship and a major motivation behind this thesis. The proposed improvements on the existing metaprogramming features, presented above in chapter 6.5, are a part of this motivation. However, by introducing a separate language processor, PScript has the ability to metaprogram both PHP and JavaScript. This ability can be employed to create novel metaprogramming features between the two main languages. PScript goal is to securely introduce metaprogramming features previously not supported by the PHP-JavaScript relationship. These features vary from macro expansion [13] to meta-level trait [18] systems. A complete list of features, examples and implementation details of metaprogramming features can be seen in chapter 7.

7 PScript: Implementation

PScript is a heterogeneous multistage language [1], [5] enabled by a novel PHP based preprocessor [2]. It is developed to interoperate with any existing PHP project, and to improve upon the existing relationship between PHP and JavaScript. This chapter introduces the implementation details of the PScript project. Each feature of PScript is presented with code examples and, when applicable, reasoning as to how each feature relates to the designated design goals. The goal is to provide an in-depth look into the functionality of PScript, while presenting the key language features it provides. [1], [2], [5]

7.1 Language Processor

The PScript preprocessor [2] remains at the core of each functionality. Compared to a traditional compiler the PScript preprocessor functions on a meta-level [1], [6]. Instead of compiling a language to executable machine code the preprocessor translates meta-level language structures to generated subject code. Therefore, the PScript preprocessor operates as a dynamic source-to-source transpiler. [1], [2]

The main goal of the preprocessor is to enable new features inside the PHP-JavaScript relationship. In order to achieve this goal, the preprocessor needs to be able to transpile variables and source code to and from both PHP and JavaScript. In the scope of this thesis, we do not implement all the features this relationship enables, but rather focus on the metaprogramming mechanism [1]. The goal of the

implementation is a secure base language that can be easily extended with more features that bridge PHP and JavaScript more tightly together. [7], [10], [15]

At a core level the implementation of the PScript preprocessor relies on PHP's ability to dynamically interpret code. As a starting point the preprocessor performs a form of lexical analysis [2], transforming the selected source code file into a purely lexical format. During the initial lexical analysis, code blocks containing purely PHP or JavaScript are separated from the PScript source code. The preprocessor then employs the PHP `eval` [11] function to interpret all native PHP into the processor's evaluation environment. Through the evaluation of existing PHP code, any predefined server-side variables and imports can be used to transpile data between the two languages. [2], [9], [13]

Example 7.1 shows an excerpt of the PScript language processor. This for-loop iterates over an array of previously detected PHP clauses. The variables are first categorized into objects, arrays and direct PScript references. The processing is then executed accordingly until all of the active variables are imported into the current evaluation environment, i.e., the PScript runtime. In the example this occurs with the usage of the aforementioned `eval` statement.

```
foreach ($php_variable_clauses[0] as $full_clause) {
    $variable_reference = $php_variable_clauses[1][
       $clause_index];
    $array_attribute = $php_variable_clauses[2][$clause_index]
        ?? null;
    $object_attribute = $php_variable_clauses[3][$clause_index
       ] ?? null;
    $value = $php_variable_clauses[4][$clause_index];

    $variable_name = str_replace('$', '', $variable_reference)
       ;

    $object_reference = '';
    if ($array_attribute) {
        $object_reference = "['{$array_attribute}']";
    }
    else if ($object_attribute) {
        $object_reference = "->{$array_attribute}";
    }

    if (str_contains($full_clause, ' client ')) {
        $client_reference = trim(str_replace('client', '',
```

```
            $value));
        $parsed_script = str_replace($full_clause, "",
            $parsed_script);

        $full_clause = $variable_reference . $object_reference
            .
        ' = PScriptVar::reference("' . $client_reference . '")
            ;';
    }

    // Evaluate variable to local scope
    eval(self::EVAL_NAMESPACE . $full_clause);
}
```

Listing 7.1: PScript: Preprocessor's usage of the eval function.

Once the language sources are separated and the dynamic variables evaluated, the preprocessor focuses on transpiling the novel PScript syntax. At the core of the transpiling process remains analysis akin to a fully fledged compilation process. First a semantic analysis [2] is executed in order to detect each PScript specific keyword. The keywords are thereafter connected with their representing logic and then expanded upon. The process effectively mimics a methodology similar to macro expansion [13], as presented in chapter 3.1.1. The transpiling logic is then applied to each of the two languages, creating a native PHP file.

Example 7.2 demonstrates a part of the semantic analysis executed during the PScript runtime. The excerpt shows a piece of the PScript preprocessor tasked with detecting inline expressions. The script first looks for the PScript syntax dedicated to expression injections, i.e., the `$[]` syntax. Valid expressions are then validated and saved inside the PScript context. The context is used for parsing and, in turn, injecting the variable into its designated place inside a client block. [19], [20]

```
// Parse all inline PHP expression injections
$inline_expression_pattern = '/(\$\s*\[\s*)([^]]+)(\s*\])/';
preg_match_all($inline_expression_pattern, $parsed_block,
    $inline_expressions);

foreach ($inline_expressions[2] ?? [] as $expression) {
    $parsed_expression = trim($expression);

    $tmp_variable_name = $this->get_hygienic_name('tmp');
    if (!str_ends_with($parsed_expression, ';')) {
        $parsed_expression = $parsed_expression . ';';
```

```
    }

    $this->context->set($tmp_variable_name, $parsed_expression
        );
    eval(self::EVAL_NAMESPACE . "$" . $tmp_variable_name . " =
        " . $parsed_expression);

    $js_value = $this->convert_variable($$tmp_variable_name);
    $parsed_block = preg_replace(
        $inline_expression_pattern,
        $js_value,
        $parsed_block,
        1
    );
}
```

Listing 7.2: PScript preprocessor semantic analysis.

During the interpretation process, the preprocessor ignores intermediate code generation [2] and semantic analysis of the PHP and JavaScript sources. This occurs due to the preprocessor being able to externalise semantic analysis to the subsequent language processors hosted by the two source languages. Effectively the preprocessor relies on the PHP and JavaScript interpreters to produce valid error messages and syntax validation from the transpiled PScript source. Similarly the processor lacks the need for separate intermediate language representations, as the PScript preprocessor's target language is native PHP. Therefore, the parsed subject code already represents, not intermediate, but rather the final language format, especially when compared to, e.g., direct machine code compilation. [2], [7], [10]

Example 7.3 demonstrates a minimal PScript variable transfer. The PHP variable `welcome` is transferred incorrectly through PScript as the variable `not_welcome`. An error occurs that is reported by the PHP runtime. The runtime correctly acknowledges the error to be a result of the variable not being present in the PScript context.

```
<?pscript
    $welcome = "Welcome to PScript!";
?>

client {
    console.log($not_welcome);
```

```
}
?>

Fatal error: Uncaught Exception: Variable 'not_welcome' not
   found in context in /var/www/pscript/processor/PScript.php
   :316
```

Listing 7.3: PScript error demonstration.

7.2 Language Features

The major motivation behind PScript is the extension of the PHP-JavaScript metaprogramming relationship [1]. This chapter describes in-depth all the new features and improvements provided by PScript. Each feature is accompanied by code examples, which demonstrate the language's features before and after the PScript transpilation process. [1], [6], [10]

7.2.1 Scoping

Scoping refers to the issue of separating language sources inside a homogeneous lexical environment [1], [7], [19]. A PHP file, for instance, combines the source codes of both PHP, JavaScript and HTML. PScript improves upon the native PHP lexical environment by implementing custom scoping for JavaScript code, instead of relying on the HTML alternative.

```
<?php
    // Demonstrative PHP block
    $php_variable = "Hello world!";
?>

client {
    // Demonstrative JavaScript block
    console.log("Hello world JavaScript!");
}
```

Listing 7.4: PScript client scoping.

Example 7.4 demonstrates the syntax PScript provides for creating a client side

JavaScript scope. As demonstrated by the example, PScript defines a JavaScript scope with the `client` keyword. The syntax for the client-side scoping is designed to resemble native PHP code instead of the native HTML alternative. Additionally the scope is defined with the client keyword reminding the developer of the scope's eventual purpose.

```
<?php
    // Demonstrative PHP block
    $php_variable = "Hello world!";
?>

<script "id"="pscript-block-0">
    // Demonstrative JavaScript block
    console.log("Hello world JavaScript!");
</script>
```

Listing 7.5: PScript client scoping — transpiled.

Example 7.5 demonstrates the transpiled version of 7.4. One can notice that the HTML `<script>` block is generated into the PHP source file. This is done in order to provide cross-compatibility with the PHP interpreter.

7.2.2 Hygienic Variable Transfer

As described in chapter 6.5, PHP does not support complete variable transfer. The PScript language processor mends this issue through enabling hygienic and dynamic variable transfers [1], [7], [13] between PHP and JavaScript.

```
<?php
    // PHP Variable definitions
    $print_clause = true;
    $print_text = "Hello world!";
?>

client {
    // PScript variable transfer
    const print_clause = $print_clause;
    const print_text = $print_text;

    if (print_clause) {
        console.log(print_text);
    }
}
```

Listing 7.6: PScript Hygienic variable transfer.

Example 7.6 demonstrates a PHP-to-JavaScript variable transfer trough PScript. PScript allows native PHP variables references to be used directly inside client blocks using the native `$variable-name` reference syntax. PScript therefore enables PHP variables to be freely mixed with native JavaScript variables and code.

```
<?php
    // PHP Variable definitions
    $print_clause = true;
    $print_text = "Hello world!";
?>

<script "id"="pscript-block-0">
    const print_text_k00M843G = "Hello world!";
    const print_clause_tW1P08E0 = true;

    // PScript variable transfer
    const print_clause = print_clause_tW1P08E0;
    const print_text = print_text_k00M843G;

    if (print_clause) {
        console.log(print_text);
    }
</script>
```

Listing 7.7: PScript Hygienic variable transfer — transpiled

In example 7.7 one can see how PScript variables are eventually generated inside the transpiled JavaScript code. PScript creates a hygienic constant variable at the beginning of the JavaScript code block containing the transpiled PHP value. It then replaces references to the original variable with references to the dynamically generated one. Each value is transpiled separately depending on the contents of the original PHP variable. During the automated transpilation process PScript can ensure variable transfers with both type safety and type conversions. Additionally PScript avoids name collisions by hygienically naming each of the generated constant variables. The hygiene is added to the variable references by appending a randomised string to the name of the generated constant variable.

7.2.3 Variable Cross-References

Cross language variable references refer to a multistage programming feature where subject stages can directly reference variables in subsequent language stages [5], [7], [20]. In addition to allowing JavaScript to directly use PHP variables, PScript enables PHP variables to contain direct references to JavaScript functions and variables.

```
<?pscript
    $double_func_reference = client double;
    $random_amount_reference = client random_amount_js;
    $random_amount_php = rand(1, 10);
?>

client  {
    const random_amount_js = $random_amount_php;
    function double(amount) {
        return amount * 2;
    }
}

client {
    const value = $double_func_reference(
        $random_amount_reference
    );
    console.log(value);
}
```

Listing 7.8: PScript Cross Language Variable References.

Example 7.8 demonstrates how references to JavaScript entities are passed to PHP variables. The variable `double_func_reference` creates a reference to the JavaScript function `double` by employing the PScript `client` keyword. In a similar manner the `random_amount_reference` references the client variable `random_amount_js`. The value of variable `random_amount_js`, however, depends on the value of the `random_amount_php`, the value of which is generated during the PHP runtime. Therefore, the value to the reference is generated dynamically during the PHP runtime and then transferred to the JavaScript variable. Example 7.8, therefore, also demonstrates that PScript allows variables to cross both runtime and language borders. [20]

```
<?php
    $random_amount_php = rand(1, 10);
?>

<script "id"="pscript-block-0">
    const random_amount_php_Ke14z74Z = 1;
    const random_amount_js = random_amount_php_Ke14z74Z;

    function double(amount) {
        return amount * 2;
    }
</script>

<script "id"="pscript-block-1">
    const random_amount_reference_l9f988ND = random_amount_js;
    const double_func_reference_e7484v35 = double;

    const value = double_func_reference_e7484v35(
        random_amount_reference_l9f988ND
    );

    console.log(value);
</script>
```

Listing 7.9: PScript Cross Language Variable References — transpiled.

Example 7.9 demonstrates the aftermath of the previously defined cross language references. One can notice how the `random_amount_php` variable has now been resolved and transferred to the JavaScript runtime, while the reference of `random_amount_js` now refers to the aforementioned generated variable. The other two variables `random_amount_js` and `double_func_reference` instead have now resolved their references into the JavaScript environment. One can notice how the hygienic reference variables combine with the cross language references allowing for the generations of the `value` variable. The `value` variable contains the combined value of both the JavaScript `double` function and the randomised value generated during the PHP runtime. Additionally, one can notice that the PHP client references are removed after they are resolved, allowing the PHP block to remain safe for the native PHP interpreter.

7.2.4 Conditional compilation

Conditional evaluation [5], [7] multistage programming strategy [10] for a language stage to affect the compilation of the latter subject stages. In the case of PScript the meta-level stage of PScript is able to provide the PHP runtime with a conditional compilation mechanism. By splicing PHP variables into the JavaScript subject stage, PScript can achieve conditional compilation in a secure and usable manner.

```
<?pscript

$hello_world1 = "Hello world 1!";
$hello_world2 = "Hello world 2!";

$show_hello1 = true;
$show_hello2 = false;

?>

client {
    if ($show_hello1) {
        console.log($hello_world1);
    }
}

client {
    if ($show_hello2) {
        console.log($hello_world2);
    }
}
```

Listing 7.10: PScript Conditional Evaluation.

Example 7.10, demonstrates how transferred PHP variables can be used for conditional compilation inside the JavaScript subject stage. Here the PHP variables named `show_hello1-2` are used directly inside the JavaScript code through hygienic variable transfer. The variable `show_hello1` enables compilation while the variable `show_hello2` dynamically revokes it.

```
<?php
$hello_world1 = "Hello world 1!";
$hello_world2 = "Hello world 2!";

$show_hello1 = true;
$show_hello2 = false;
?>
```

```
<script id="pscript-block-0">
    const hello_world1_sm65S0Lw = "Hello world 1!";
    const show_hello1_m242a801 = true;

    if (show_hello1_m242a801) {
        console.log(hello_world1_sm65S0Lw);
    }
</script>

<script id="pscript-block-1">
    const hello_world2_dMkYDQ2t = "Hello world 2!";
    const show_hello2_beb3467Y = false;

    if (show_hello2_beb3467Y) {
        console.log(hello_world2_dMkYDQ2t);
    }
</script>
```

Listing 7.11: PScript: Conditional Evaluation — transpiled.

Example 7.11 demonstrates the result of the conditional compilation. The transferred PHP variables are evaluated during the PScript runtime, leaving behind the boolean statements contained within them. The JavaScript interpreter will interpret the boolean statements and if-clauses as conditionally compiled code blocks. This approach can also be extended to conditionally compile full code blocks or even to conditionally change the value of variables on an individual basis. [7], [19]

7.2.5 Expression Injection

Expression injection is the next natural step-up from subject code variable injection [7], [21]. PScript implements expression injection through a native quasi-quotation [22] syntax. Quasi-quotations `$[]` allow for the evaluation of meta-level expression directly inside a subject stage environment [22]. PScript's quasi-quotation syntax follows the styling of native PHP syntax by using the familiar `$` and `[]` symbols for the creation of the quasi-quotation scope.

```
<?pscript

$hello = "Hello ";
$world = "world!";

$print1 = false;
```

```
$print2 = true;

?>

client {
    const js_hello_world = $[$hello . $world];
    const js_hello_world2 = js_hello_world + " - 2";

    if ($[$print1 && $print2]) {
        console.log(js_hello_world);
    }

    if ($[$print1 || $print2]) {
        console.log(js_hello_world2);
    }
}
```

Listing 7.12: PScript Expression Injection.

Example 7.12 demonstrates the usage of PScript's quasi-quotation methodology. The variable `js_hello_world` contains the value of a PScript expression injection. Inside the quasi-quotation PScript evaluates and then combines the values of two separate PHP expression that are then injected into the aforementioned variable. The latter quasi-quotations demonstrate how expression injection can also be used directly inside native JavaScript syntax.

```
<?php

$hello = "Hello ";
$world = "world!";

$print1 = false;
$print2 = true;

?>

<script "id"="pscript-block-0">

    const js_hello_world = "Hello world!";
    const js_hello_world2 = js_hello_world + " - 2";

    if (false) {
        console.log(js_hello_world);
    }

    if (true) {
        console.log(js_hello_world2);
    }

</script>
```

Listing 7.13: PScript Expression Injection — transpiled.

Example 7.13 demonstrates how the PScript quasi-quotations are evaluated during the PScript runtime. In the transpiled result the variable `js_hello_world2` evaluates into `"Hello world!  2"`. Similarly the boolean expressions injections inside the if statements are now evaluated and provide conditional compilation inside the JavaScript runtime.

7.2.6 Namespaces

Namespaces allow for the organization of variables into well defined sub scopes inside the general namespace of the main program. On a meta-level, metaprogramming solutions like PScript can provide dynamic namespaces through injecting meta-level variables directly into the underlying subject stages [7], [20]. Dynamic namespaces, in turn, allow for the avoidance of name collisions while providing safe namespaces for transferring meta-level variables and references. [9], [21]

```
<?pscript
    $namespace1 = [];
    $namespace2 = [];

    $namespace1['hello_world'] = client hello1;
    $namespace2['hello_world'] = client hello2;

    $namespace1['print'] = false;
    $namespace2['print'] = true;
?>

client  {
    function hello1() {
        if ($namespace1['print']) {
            console.log("Hello world 1!");
        }
    }

    function hello2() {
        if ($namespace2['print']) {
            console.log("Hello world 2!");
        }
    }
}
```

```
client {
    $namespace1['hello_world']();
}

client {
    $namespace2['hello_world']();
}
```

Listing 7.14: PScript Dynamic Namespaces.

Example 7.14 demonstrates the usage of meta-level namespaces through PScript. PScript provides the functionality through transferring native PHP associative arrays into active client scopes. For instance in 7.14 the separate `namespace1-2` variables are used to create two meta-level namespaces. The namespaces are employed to store references to client functions and to meta-level PHP variables. The references are then used from inside the subject level client scopes creating two separate namespaces.

```
<?php
    $namespace1 = [];
    $namespace2 = [];

    $namespace1['print'] = false;
    $namespace2['print'] = true;
?>

<script id="pscript-block-0">
    const namespace2_print_zx96X73E = true;
    const namespace1_print_OJ10059N = false;

    function hello1() {
        if (namespace1_print_OJ10059N) {
            console.log("Hello world 1!");
        }
    }

    function hello2() {
        if (namespace2_print_zx96X73E) {
            console.log("Hello world 2!");
        }
    }
</script>

<script id="pscript-block-1">
    const namespace1_hello_world_kb15r2V5 = hello1;
    namespace1_hello_world_kb15r2V5();
</script>

<script id="pscript-block-2">
    const namespace2_hello_world_IG82wb8H = hello2;
```

```
    namespace2_hello_world_IG82wb8H();
</script>
```

Listing 7.15: PScript Dynamic Namespaces — transpiled.

Example 7.15 shows how the dynamic namespaces are transpiled into subject level code. The references to each namespace-variable are first hygienically generated into new variables and thereafter resolved locally inside the subject code blocks. Effectively this creates a hygienic namespace inside the subject code while allowing the meta-stage to group variables and references in a convenient manner.

7.2.7 Templates & Specialization

Templates [17] are a way for meta-level stages to generate subject level code dynamically. Templates are especially useful in dynamically generating subject level objects based on meta-level requirements. PScript provides template generation through conditional compilation and through injecting client code blocks into subject-level code.

Template functionality can also be used for the specialisation of subject-level code [9], [17]. As meta-level code effectively controls what code is included inside a generated subject class, meta-level structures can be used to specialise subject code. PScript enables specialization through the injection and evaluation of meta-level PHP variables inside generated templates.

```
<?pscript
    function create_animal($animal_type) {
        if ($animal_type == "cat") {
            return client {
                class Animal {
                    who_am_i() {
                        return "Cat";
                    }

                    say_hello() {
                        console.log("Meow");
                    }
                }
```

```
            }
        }
        else if ($animal_type == "dog") {
            return client {
                class Animal {
                    who_am_i() {
                        return "Dog";
                    }

                    say_hello() {
                        console.log("Woof");
                    }
                }
            }
        }
    }
?>

client {
    $[create_animal("dog")]

    const animal = new Animal();
    console.log("Found a " + animal.who_am_i());
    animal.say_hello();
}
```

Listing 7.16: PScript Template.

Example 7.16 demonstrates how the PScript methodologies of cross-language references, client scoping and quasi-quotations [22] are used to provide meta-level templating functionality. The function `create_animal` acts as a meta-level template for generating different specialisations of the JavaScript class `Animal`. The provided function parameter `animal_type` determines which of the two animal classes are generated, i.e., a cat or a dog. In the quasi-quotation block one can notice that the expression currently evaluates to the latter and generates a JavaScript class `Animal`, with the `dog` specialisation.

```
<?php
    function create_animal($animal_type) {
        if ($animal_type == "cat") {
            return PScriptBlock::create("0");
        }
        else if ($animal_type == "dog") {
            return PScriptBlock::create("1");
        }
    }
?>
```

```
<script id="pscript-block-0">
    class Animal { who_am_i() { return "Dog"; } say_hello() {
       console.log("Woof"); } }
    const animal = new Animal();
    console.log("Found a " + animal.who_am_i());
    animal.say_hello();
</script>
```

Listing 7.17: PScript Dynamic Template — transpiled.

Example 7.17 demonstrates how the specialised animal class is transpiled into native JavaScript code. One can notice that the compiled class is injected into the transpiled code block in the place of the meta-level quasi-quotation. The native JavaScript code in turn demonstrates how the metaprogrammed Animal class can be natively referenced inside the JavaScript runtime. For instance, the `say_hello` function will now print out `Woof` due to the specialisation that occurred during the meta-level PScript runtime.

7.2.8 Traits

Traits [18] enable the injection of predefined attributes into subject-level classes based on meta-level keywords. Usually these keywords are embedded within a language itself. In the multistage language created by PScript the keywords can be provided by the meta-level PScript stage [1], [5]. PScript provides trait functionality much in the same way as demonstrated previously with templating, i.e., PScript injects generated subject level code directly into desired places inside the client scope using the quasi-quotation feature [22].

```
<?pscript
    function say_hello_trait() {
        return client {
            say_hello() {
                console.log(
                    this.who_am_i() +
                    " says - " +
                    this.what_sound() +
                    "!"
                );
            }
```

```
        }
    }
?>

client {
    class Animal {
        $[say_hello_trait()]

        who_am_i() {
            return "Dog";
        }

        what_sound() {
            return "Woof";
        }
    }

    const animal = new Animal();
    animal.say_hello();
}
```

Listing 7.18: PScript Traits.

Example 7.18 demonstrates how traits [18] are employed inside the PScript environment. In this case the function `say_hello_trait` is used to define a client block containing the desired attribute. Here the generated attribute `say_hello` enhances the `Animal` class with a function attribute `say_hello`. The attribute itself simply prints out the results of two separate variable attributes. The meta-level code does not have access to these variables, instead the access is achieved through the PScript preprocessor transpilation process.

```
<?php
    function say_hello_trait() {
        return PScriptBlock::create("0");
    }
?>

<script id="pscript-block-0">
class Animal {
    say_hello() { console.log( this.who_am_i() + " says   " +
       this.what_sound() + "!" ); }

    who_am_i() {
        return "Dog";
    }

    what_sound() {
        return "Woof";
    }
```

```
}

const animal = new Animal();
animal.say_hello();
</script>
```

Listing 7.19: PScript Traits — transpiled.

Example 7.19 demonstrates how PScript traits affect the transpiled subject code. Here the quasi-quotation reference to the `say_hello` trait is replaced with the predefined trait attribute. The generated attribute function now has access to the other native attributes inside the subject class. Much like the template system, the original PHP-PScript trait template is sanitized into a valid PHP reference.

The attribute provided by the meta-level trait could be made more general by removing the references to specific class attributes. This would allow the trait to function in specialization of any generic subject class. By overriding existing class attributes or implementing interfaces, the PScript trait system could be extended to cover functionality similar to that of aspects and aspect weaving.

8 PScript: Demonstration

Chapter 7 describes the plethora of improvements PScript introduces into the PHP-JavaScript relationship. The demonstrations and examples were chosen to introduce each feature in a simple context. This chapter demonstrates the use, and benefits, of PScript in solving a real-world problem. All of the code required for this demonstration is available at the project's Github page. [23]

8.1 Problem description

Google remains a major player in providing accurate user tracking and site usage analytics to active web applications. Google's tracking solutions were deemed to be so effective that as of 2024 their usage in web applications were limited by regulation. Modern web applications are now required to collect and store user consent for using their data for marketing or analytics of any web application. Therefore, web applications often collect these required consents with so-called 'Cookie banners'. The banners, in turn, provide the user with controls for deciding on what tracking the user consents to.

The problem this demonstration solves using PScript occurs on the server side: asking user consent only when necessary. A returning or logged in customer will already have given their consent when opening an arbitrary page on the web application. Therefore, the web application should not ask for user consents continuously when the user continues to explore the site.

The problem is further complicated via the usage of the commonly used tracking pixels [24] provided by Google and other third parties. Often these tracking pixels contain code that needs to be run directly on the client side. However, in order to correctly configure them from the start, one needs to pass the server-side consent data onto the client-side tracking pixels. Furthermore, some pixels require more complex configurations related to the user's consents, as they can, e.g., operate using anonymous data. [24]–[26] prerequisites

8.2 Environment and Prerequisites

The following demonstration is transpiled using the PHP [3] version 8.2 [11]. The demonstration occurs inside a docker [27] container routed using a Nginx [28] web server. As prerequisites, the demonstration will present two JavaScript classes `GoogleAnalytics` and `GoogleAds`. These classes are meant to act as wrapper classes for active pixel code provided by an actual Google service. In the case of this demonstration the classes are just dummy versions meant to log the different settings given to them during the client side runtime. Additionally, the demonstration requires a user entity that contains the saved user consents. In this demonstration the user is always dynamically generated dummy user, but the data could also be fetched directly from an active database. [24]–[26]

8.3 Implementation

Example 8.1 demonstrates the raw PScript source required to solve the described problem. As mentioned in chapter 7, PScript can import external PHP files into the scope of the PScript runtime. In the case of this demonstration, the `require_once` statement imports the `finale.php` file, which contains the required prerequisite classes and functions described above 8.2. After the required prerequisite functions

are imported, the script sets up additional variables required for its functionality. The variables contain demonstrative user ids required for the pixel code to function and, more importantly, an array containing the fetched user consents.

After the script perquisites are set up, the script moves to define the actual functionality. The PScript functions `google_analytics_tracker` and `google_ads_tracker` specialize the templates required for the google tracking pixels. Both of the functions set up their respective trackers by initiating the wrapper object, defining the client id and enabling the tracker.

However, the two functions act differently based on the defined user consents. For instance, the `google_ads_tracker` is conditionally compiled based on the user's marketing consent. If the user gives consent for marketing based tracking, the server-side PScript code, in turn, enables the tracker. Similarly the analytics tracker respects the user's decision on allowing their data to be used for analytics tracking. However, the Google analytics pixel can additionally function in an anonymized data mode that needs to be enabled separately on the client side. Therefore, the `google_analytics_tracker` is conditionally generated in a different manner, using the functionality provided by PScript. If the user revokes their analytics consent, the client-side pixel code will be used to enable anonymization before the tracker is enabled.

The `user_consents_template` function is used to specialize the client-side `ConsentManager` class. The `ConsentManager` can be used to move the user's consent information to the client side. PScript is, in turn, used to metaprogram the class template, effectively filling it with the user consents fetched from the database. This approach greatly improves efficiency of transferring data between the server and client side, as otherwise such data transfer would have to happen using native client-to-server API interfaces. Regarding security, the user's consent data can be considered insensitive as it only reflects against the user's actions happening inside

the client browser.

Finally the solution defines a `client` scope that brings all of the metaprogramming functionalities together. In the final `client` block PScript's expression injection mechanism is used to import all of the specialized pixel trackers into the client-side code. The block also demonstrates how the templated `ConsentManager` class can be directly referenced in the client side code. The generated class now contains the dynamically transferred consent values, which can be used to populate the coinciding client side cookies.

```
<?pscript
    require_once( ROOT_PATH . '/public/demo/finale.php');

    $user = get_current_active_user();

    $google_ads_id = GOOGLE_ADS_ID;
    $google_analytics_id = GOOGLE_ANALYTICS_ID;
    $user_id = $user->id;

    $user_consents = [];
    $user_consents['necessary'] = $user->consents->necessary;
    $user_consents['analytics'] = $user->consents->analytics;
    $user_consents['preferences'] = $user->consents->
       preferences;
    $user_consents['marketing'] = $user->consents->marketing;

    function google_analytics_tracker($google_analytics_id,
       $user_consents) {
        return client {
            const google_analytics = new GoogleAnalytics();
            if (!$user_consents["analytics"]) {
                google_analytics.anonymize();
            }
            google_analytics.set_source_id(
               $google_analytics_id);
            google_analytics.enable();
        }
    }

    function google_ads_tracker($google_analytics_id,
       $user_consents) {
        return client {
            if ($user_consents["marketing"]) {
                const google_ads = new GoogleAds();
                google_ads.set_id($google_ads_id);
                google_ads.enable();
            }
        }
    }

    function user_consents_template($user_consents) {
```

```
        return client {
            class ConsentManager {
                #user_consents = {
                    necessary: $user_consents["necessary"],
                    analytics: $user_consents["analytics"],
                    preferences: $user_consents["preferences"
                       ],
                    marketing: $user_consents["marketing"]
                };

                get(consent) {
                    return this.#user_consents[consent];
                }

                get_all() {
                    return this.#user_consents;
                }
            }
        }
    }
?>

client {
    $[user_consents_template($user_consents);];
    const consent_manager = new ConsentManager();

    $[google_analytics_tracker($google_analytics_id,
       $user_consents);];

    $[google_ads_tracker($google_ads_id, $user_consents);];

    for (const [key, value] of Object.entries(consent_manager.
       get_all())) {
        const cookie = "consent_" + key + "=" + value;
        document.cookie = cookie;
        console.log(cookie);
    }
}
```

Listing 8.1: PScript: Google Analytics Demo.

Example 8.2 shows the transpiled native PHP and JavaScript blocks. The PScript functions have been replaced with native PHP references, only leaving behind the blank function templates. The results of the functions have already been transpiled into the main `client` block scope. For instance, the tracking pixel code blocks are injected to their respective positions based on the meta-level quasi-quotations. The metaprogrammed user consent variables now reference the hygienic versions generated by the PScript runtime.

```
<?php
```

```
    require_once( ROOT_PATH . '/public/demo/finale.php');

    $user = get_current_active_user();

    $google_ads_id = GOOGLE_ADS_ID;
    $google_analytics_id = GOOGLE_ANALYTICS_ID;
    $user_id = $user->id;

    $user_consents = [];
    $user_consents['necessary'] = $user->consents->necessary;
    $user_consents['analytics'] = $user->consents->analytics;
    $user_consents['preferences'] = $user->consents->
       preferences;
    $user_consents['marketing'] = $user->consents->marketing;

    function google_analytics_tracker($google_analytics_id,
       $user_consents) {
        return PScriptBlock::create("0");
    }

    function google_ads_tracker($google_analytics_id,
       $user_consents) {
        return PScriptBlock::create("1");
    }

    function user_consents_template($user_consents) {
        return PScriptBlock::create("2");
    }
?>

<script id="pscript-block-0">
    const google_ads_id_S30498M2 = "G-ADS-1234";
    const user_consents_marketing_lr87yh6N = false;
    const google_analytics_id_mV9C9j3n = "G-ANALYTICS-1234";
    const user_consents_analytics_d48Qt760 = true;
    const user_consents_marketing_S31xwHh8 = false;
    const user_consents_preferences_s633H95H = true;
    const user_consents_analytics_VY1ca9i6 = true;
    const user_consents_necessary_n8k7A65O = true;

    class ConsentManager { #user_consents = { necessary:
       user_consents_necessary_n8k7A65O, analytics:
       user_consents_analytics_VY1ca9i6, preferences:
       user_consents_preferences_s633H95H, marketing:
       user_consents_marketing_S31xwHh8 }; get(consent) {
       return this.#user_consents[consent]; } get_all() {
       return this.#user_consents; } } ;
    const consent_manager = new ConsentManager();

    const google_analytics = new GoogleAnalytics();
    if (!user_consents_analytics_d48Qt760) {
        google_analytics.anonymize();
    }
    google_analytics.set_source_id(
       google_analytics_id_mV9C9j3n);
    google_analytics.enable();

    if (user_consents_marketing_lr87yh6N) {
        const google_ads = new GoogleAds();
        google_ads.set_id(google_ads_id_S30498M2);
```

```
        google_ads.enable();
    }

    for (const [key, value] of Object.entries(consent_manager.
       get_all())) {
        const cookie = "consent_" + key + "=" + value;
        document.cookie = cookie;
        console.log(cookie);
    }
</script>
```

Listing 8.2: PScript: Google Analytics Demo — Transpiled.

9 Discussion

Chapter 7 demonstrates the capabilities enabled by the PScript preprocessor. In the aggregate the described features seek to enable the design goals described in chapter 6. At its core the PScript language processor is made to extend the existing relationship between PHP and JavaScript.

Before the relationship could be improved upon, it required a secure structure on top of which to build the improvements. PScript provides this secure structure through expanding the existing relationship into a multistage language. The multistage language itself consists of three separate runtimes: PScript, PHP and JavaScript. The latter two rely on the existing language interpreters of PHP and JavaScript, while the PScript runtime itself remains a pseudo-runtime inside the PHP life-cycle. The PHP runtime provides PScript with passive up-keep and an existing user-base. The PScript runtime is able to approach the existing PHP-JavaScript life-cycle in an efficient manner while ensuring its usability in existing and future solutions.

Expanding the existing relationship with an additional language step gives PScript freedom to work around the two language environments. This freedom is mainly provided through the means of the custom PScript language preprocessor. The preprocessor moves the existing PHP and JavaScript runtimes into the role of subject stages of the multistage language structure. I.e., the PScript runtime becomes the initial meta-stage of the multistage relationship. By moving the PHP runtime into a

subsequent subject stage, the PScript runtime is able to metaprogram the relationship it shares with JavaScript. Hence, PScript is given the ability to metaprogram both JavaScript and PHP, allowing it to improve upon the metaprogrammability of the relationship itself. This approach allows the language processor to improve upon the existing relationship according to all the predefined design goals.

One of these improvements was to deal with the poor usability of the relationship, as discussed in chapter 6.2. Usability remains a key issue at the PHP JavaScript relationship, so much so that the industry limits the usage of PHP's existing language features. Therefore, bringing improvements in the usability vertical is necessary for enabling further use of the relationship.

The main ways PScript tackles the issue of poor usability is to make sure the creation of a multistage language environment does not disturb the existing language environments. For instance, the way the PScript runtime is invoked without disturbing the existing PHP runtime became a key guideline in PScript's design. As demonstrated in chapter 6, PScript introduces a new syntax for invoking the PScript runtime, i.e., the `PScript::require()` statement. The usage of the statement `PScript::require()` well summarised PScript's overall dedication to usability inside existing projects. The statement itself remains native to the PHP runtime, which is crucial for extending the user-base of the project. At the same time the native appearance is meant to reduce the overall on-boarding required for adopting PScript in a project. Additionally, all novel syntax implementations provided by PScript follow the same syntax styling as PHP itself. Therefore PScript syntax should feel familiar to existing developers, while also remaining consistent inside the PHP environment. A conscious decision was also made to reduce the amount of new syntax, in order to limit the language processor's impact on existing code bases' usability.

PScript's efforts to improve the usability of the multistage-language can also be

argued to positively affect the clarity of the solution. For instance, invoking the PScript runtime always happens through the native `PScript` statement, making it clear that the clause affects the PScript runtime. Additionally, PScript's features such as hygienic variable transfer also respect the notion of clarity. I.e., during a cross language variable transfer all transferred variables maintain their respected naming regardless of increased hygiene.

On the other hand, PScript has to deal with the increased complexity of a multi-stage language, which is bound to affect both the clarity and usability of the solution. PScript seeks to maintain clarity inside the multistage environment by making each of the separate runtime borders as clear as possible. Instead of requiring the usage of HTML syntax, PScript provides a clear replacement for generating JavaScript scopes that both adheres to PHP syntax styling and provides a clear separation between the client and server side languages.

Additionally, PScript addresses clarity of the language environment through the usage of the `client` keyword. The `client` keyword remains a consistent reminder for separating between the server and client runtimes, designed to remind the user that the features and scopes created by PScript address the underlying client side code of JavaScript. The reminder is key in increasing both the security and clarity of the server to client side relationship. PScript also improves upon the safety of the multi-language environment by first parsing, transpiling and then validating each transferred variables. This validation guarantees that types are transpiled correctly. The feature could be extended to support, e.g., the automated detection of sensitive info.

PScript's efficiency considerations also extend to maintaining the efficiency of the PHP runtime. A key consideration is PScript's ability to cache previously parsed source files. Therefore, PScript's load on an existing PHP runtime is minuscule at best. Additionally, the PScript preprocessor's ability to both cache and dynamically

generate source files means that the solution remains usable even in an active build pipeline.

PScript can also be argued to improve the efficiency of the PHP-JavaScript language environment itself. For instance, PScript introduces new syntax for dealing with existing problems, e.g., cross language variable references. In the same manner, each feature described in chapter 7 is designed to enhance the existing relationship, while also improving the efficiency thereof. For instance, instead of managing JavaScript code through file inclusion, PScript provides a native way to conditionally compile JavaScript through variable and expression injections. Similarly, the ability to transfer data between the server and client side code significantly improves the efficiency of specialising client-side behaviours by using the server-side runtime. Ultimately all of these improvements enhance the experience of both developers and users of the multistage solution.

Finally PScript's main goal is to improve the existing PHP-JavaScript relationship, not just fix its current issues. PScript enables such improvements through the plethora of novel metaprogramming features demonstrated in 7. At the core of each feature is the novel language processor implementing a preprocessor based compiler pattern. The PScript preprocessor enables new features, such as, template specialization and dynamic code generation, previously not available to either PHP or JavaScript. The key with each introduced feature is that they extend the existing relationship making its usage feel both natural and secure. Therefore, the language processor enables completely new approaches to dealing with existing issues inside the multistage environment.

In this manner the PScript preprocessor not only improves upon the existing relationship, but also enables features not present in the native languages. For instance PScript enables PHP to store client side code and references inside native variables. PScript extends these references to introduce novel metaprogramming

methodologies, e.g., client code templates and specialisation thereof. As a result PScript, is able to provide even the client-side JavaScript with considerable features, such as the meta-level trait system.

All in all one can argue that all the features of PScript together improve the PHP-JavaScript both through securing the existing language environments and providing novel features. In the scope of this thesis, PScript remains a proof of concept. The multistage-language has many improvements that could be explored in future research. These include, e.g., further securing the relationship through managing sensitive server-side information through the PScript language processor. The secure base of PScript also enables for improvements in introducing further metaprogramming features. For instance, the introduction of dynamic macro systems is a possible vertical for expanding the metaprogrammability of the solution.

10 Conclusion

JavaScript and PHP share a server to client relationship. The relationship provides a vertical for metaprogramming between the two languages. This thesis argues that this relationship remains incomplete, and introduces the concept of a new language preprocessor PScript. The design of PScript was shown to improve the usability, efficiency and security of the PHP-JavaScript relationship through novel language features, such as dynamic namespaces and cross language variable transfers. On top of this, the new metaprogramming features were shown to further enhance the usability and clarity of the language relationship. Ultimately even at its current prototype stage PScript improves the existing relationship both in theory and in practice. PScript prototype is a secure base for a novel language processor, on top of which further research can be built. PScript is an introduction into secure metaprogramming and it can be employed to enhance the efficiency and security of complex real-world programs.

References

[1] Y. Lilis and A. Savidis, "A survey of metaprogramming languages", *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019, ISSN: 0360-0300. DOI: 10.1145/3354584. [Online]. Available: https://doi.org/10.1145/3354584.

[2] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

[3] *PHP: Hypertext Preprocessor — php.net*, https://www.php.net/, [Accessed 16-03-2024].

[4] M. MozDevNet, *Javascript language overview - JavaScript: Mdn*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview.

[5] W. Taha, "A gentle introduction to multi-stage programming", in *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, Springer, 2004, pp. 30–50.

[6] R. Damaševičius and V. Štuikys, "Taxonomy of the fundamental concepts of metaprogramming", *Information Technology and Control*, vol. 37, no. 2, 2008.

[7] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: A multi-stage language for high-performance computing", in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 105–116.

[8] C. Calcagno, W. Taha, L. Huang, and X. Leroy, "Implementing multi-stage languages using asts, gensym, and reflection", in *International Conference on Generative Programming and Component Engineering*, Springer, 2003, pp. 57–76.

[9] L. Tratt, "Compile-time meta-programming in converge.", 82wqw, Tech. Rep., 2002.

[10] J. de Oliveira Guimarães, "The Cyan language metaobject protocol", 2023.

[11] *PHP: Releases — php.net*, `https://www.php.net/releases/index.php`, [Accessed 06-07-2024].

[12] M. MozDevNet, *Javascript language overview - JavaScript: Mdn.* [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview`.

[13] E. Burmako, "Scala macros: Let our powers combine! on how rich syntax and static types work with metaprogramming", in *Proceedings of the 4th Workshop on Scala*, 2013, pp. 1–10.

[14] G. Neverov and P. Roe, "Towards a fully-reflective meta-programming language", in *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*, 2005, pp. 151–158.

[15] J. Guimarães, "Metaprogramming in Cyan", *Available at SSRN 4385794*,

[16] S. Roychoudhury, J. Gray, H. Wu, J. Zhang, and Y. Lin, "A comparative analysis of meta-programming and aspect-orientation", in *Proc. Of the 41st Annual ACM SE Conference, Savannah, GA*, 2003.

[17] Z. Porkoláb, "Functional programming with C++ template metaprograms", in *Central European Functional Programming School*, Springer, 2009, pp. 306–353.

[18] J. Reppy and A. Turon, "A foundation for trait-based metaprogramming", in *International workshop on foundations and developments of object-oriented languages*, 2006.

[19] Z. DeVito and P. Hanrahan, "The design of terra: Harnessing the best features of high-level and low-level languages", in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2015.

[20] *Terra — terralang.org*, `https://terralang.org/`, [Accessed 06-07-2024].

[21] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan, "First-class runtime generation of high-performance types using exotypes", in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 77–88.

[22] J. Wielemaker and M. Hendricks, "Why it's nice to be quoted: Quasiquoting for Prolog", *arXiv preprint arXiv:1308.3941*, 2013.

[23] L. Haapala, *GitHub - lphaap/php-script — github.com*, `https://github.com/lphaap/php-script`, [Accessed 06-07-2024].

[24] Google, *About the Google tag - Google Ads Help — support.google.com*, `https://support.google.com/google-ads/answer/11994839`, [Accessed 06-07-2024].

[25] Google, *Set up consent mode - Google Ads Help — support.google.com*, `https://support.google.com/google-ads/answer/14009635`, [Accessed 06-07-2024].

[26] Google, *Unblock Google tags when using consent mode - Tag Manager Help — support.google.com*, `https://support.google.com/tagmanager/answer/12962079`, [Accessed 06-07-2024].

[27] *Home — docker.com*, `https://www.docker.com/`, [Accessed 06-07-2024].

[28] *Nginx — nginx.org*, `https://nginx.org/en/`, [Accessed 06-07-2024].