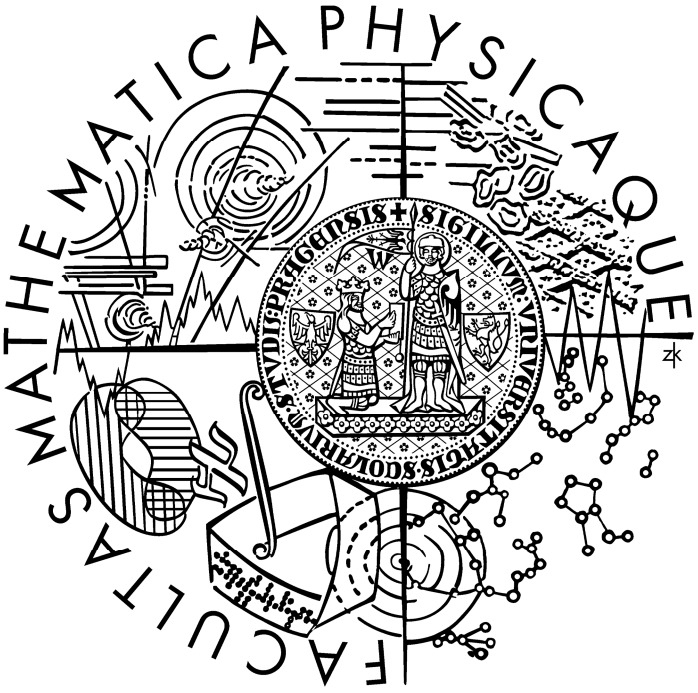
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Daniel Balaš

Advanced Optimizations in Dynamic Language Compiler

Department of Theoretical Computer Science and Mathematical Logic
Supervisor of the master thesis: RNDr. Filip Zavoral, PhD.
Study programme: Computer Science
Specialization: Theoretical Computer Science

2011

First, I would like to thank Filip Zavoral for the professional guidance and patience he gave me during writing of this thesis and especially during its finishing.

Thanks to Tomáš Petříček who gave me invaluable feedback and support during writing of this thesis. Thanks to my colleagues Miloslav Beňo and Jakub Míšek for struggling with development of Phalanger without me for some time. Thanks to my parents for helping me and supporting me throughout my studies. Thanks to Zuzana Donátová for her optimism and professionality. Thanks to Sam for being such a great kid. Thanks to Adam Abonyi and Jan Ambrož just for the fact that they are my friends.

Above all, I would like thank to my wife, Eva, for supporting me and standing beside me for five long and difficult years and for helping me to deal with my illness, which I would not be able to get over without her.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In..... date..... Signature

Název práce: *Pokročilé optimalize v překladači dynamického jazyka*

Autor: *Daniel Balaš*

Katedra / Ústav: *Katedra teoretické informatiky a matematické logiky*

Vedoucí diplomové práce: *RNDr. Filip Zavoral, PhD., Katedra softwarového inženýrství*

Abstrakt:

Dynamické programovací jazyky jsou navržené pro běh v interpretu a tím pádem provádějí značnou část typových kontrol za běhu. Oproti tomu překladače statických jazyků mohou značnou část této třídy kontrol odstranit za pomoci informací, které jsou jim známy v době překladu. Tato práce je zaměřena na zlepšení projektu Phalanger, který umožnuje překlad a běh programů implementovaných v jazyce PHP na platformě .NET, za pomoci statických programových analýz. Tyto analýzy umožní odstranit velkou část dynamických operací, které jsou nyní prováděny za běhu. V práci ukazujeme, že pečlivým navržením analýzy programového toku a následné typové analýzy jsme schopni několikanásobně zvýšit rychlost běhu jednoduchých PHP programů.

Klíčová slova: *PHP, překladač, typová analýza, Phalanger, optimalizace*

Title: *Advanced Optimizations in Dynamic Language Compiler*

Author*: Daniel Balaš*

Department / Institute: *Department of Theoretical Computer Science and Mathematical Logic*

Supervisor of the master thesis: *RNDr. Filip Zavoral, PhD., Department of Software Engineering*

Abstract:

Dynamic programming languages are designed to be interpreted and consequently perform a considerable amount of runtime type checking. In contrast, compilers for statically typed languages can remove this class of checks using information known at compile-time. This thesis aims to improve Phalanger, a compiler of PHP language for CIL, by introducing advanced static program analyses to its compilation process in order to remove some, or most, of dynamic operations that are now performed at runtime by Phalanger-compiled programs. We show that using a carefully designed control flow analysis and subsequent type analysis we are able to improve the performance of simple PHP programs several times.

Keywords: *PHP, compiler, type analysis, Phalanger, optimization*

Contents

Introduction

Dynamic languages are distinct from static languages mainly by being designed to do majority of type-related checks at runtime. Because of their simplicity, succinctness and a fast learning curve they are now very popular (for example *PHP* [**1**], *Perl* [**2**] and *Ruby* [**3**]). Many of dynamic language features that were considered strictly dynamic two decades ago are now present in mainstream static languages such as *C#* and *Java* – runtime type information, reflection, runtime code generation, late binding etc.

Perhaps the most severe problem of dynamic languages is performance – traditionally these languages were either interpreted directly or transformed to an optimized byte-code which was then interpreted by a virtual machine. Although the byte-code interpreters have tolerable performance penalty in most applications, mainly because the byte-code itself is designed to be efficiently interpreted, mentioned move to business applications marked interpreted dynamic languages as slow. Compilation of dynamic languages is achieved by:

- Transforming dynamic program to static program whenever it is possible using static analysis [**4**] [**5**] [**6**].
- Limiting or removing dynamic features that pose problem for compilation [**7**].
- Combination of mentioned approaches [**8**].

Byte-code interpreting itself is not the only performance issue – if done properly it can be less than tenfold slower than equivalent program implemented in a static language. The other source of performance problems is the late binding of various program elements, which are usually identified by their name. Generic mapping with string key using common hash table implementations is not fast enough to compete with early binding – according to our testing the computation resource cost of such access can be as much as hundredfold higher for local variables and functions. Such high overhead is present for each variable access, function call etc. making the overall performance much slower. Obvious approach to overcome this is by performing static analysis that transforms late binding to early binding. Other approach is to use map implementations that are more efficient (e.g. cache-conscious hash table [**9**] or trie [**10**]).

Our previous work [**11**] [**12**] was concerned with optimizations and general improvements of *Phalanger* [**6**] [**13**], an open-source *PHP* compiler and runtime for *CLI* [**14**]. Main goal of this project is to make *PHP* a first-class citizen among *.NET* programming languages. Phalanger refines the *PHP* language with so-called *PHP/CLR* semantics, which allow interoperability with *.NET* libraries. Additionally, *Phalanger* includes tools that make using of *PHP* programs in *.NET* language possible to a certain extent. Being a dynamic language compiler, it performs several static analyses to remove dynamic operations and transform them into static *CIL* byte code.

However, analyses performed by *Phalanger* do not consider program's flow and infer only results that are globally valid. Since the main property of dynamic languages is that semantics are based on runtime execution context, such restriction to globally valid results proves to be the major limitation of the compiler.

In our opinion, *PHP* is a good choice for dynamic language compilation research because it has several properties that make static analyses easier and their results more valuable, for example:

- Type system is nominative and consists of primitive types, multi-purpose arrays and hierarchy of classes. Thus, the type system is not dynamic per se; only the values are dynamically typed, allowing a *type analysis* to be efficient, because the set of types is usually very small.
- Once a function or method declaration is made at runtime it cannot be changed or removed later. This allows transforming late bound references to these declarations to early bound in case it was proven that the declaration will be always made before the call.

On the other hand *PHP* has several specific features that make compilation and static analysis hard with widely used techniques and led to the fact that interpreters are industry-standard for using *PHP* programs. These are for example:

- An interpreter was presumed when designing language semantics. Good example is that *PHP* library functions can access call context properties during evaluation, i.e. calling function, changing scope variables etc.
- PHP reference semantics enable multiple places (global variables, local variables, object properties and array items) sharing value. Reference operator is only difference with using non-referenced values, because dereferencing of referenced values happens implicitly.

However, most hard-to-compile dynamic features, such as *EVAL*, dynamic properties, magic members or indirect operations, are not used by *PHP* programmers often and if are, it is in most cases possible to infer the actual value and remove such operation. This fact can be taken as assumption by the compiler and such operations don't need to be more efficient than in original implementation.

Aside from the language itself, *Phalanger* implementation and *CLR* ecosystem introduce other properties with additional positive effects on analysis and compilation efficiency:

- Library functions are implemented in *C#* and are inherently typed and the typing can be used by the compiler. This provides much better starting ground for type analysis and inference.
- Call-context and other dynamic features are not used by most functions and functions using them are explicitly marked by *.NET* attributes, which enable the compiler to optimize the call by not supporting these semantics whenever possible.
- Special properties of different functions can be marked by *.NET* attributes, improving the starting point for many analyses. For example, functions that do not have side-effects can be marked with an attribute representing referential transparency.

It is important to note that *Phalanger* is not only a research project, but it is targeted at business applications as well. As mentioned before, dynamic languages were first widely used technologies for implementing Web applications and *PHP* was and still is very popular in this environment. Though nowadays most large enterprise Web applications are implemented using *ASP.NET* (mostly in *C#* and sometimes in *Visual Basic .NET* or *Visual F#*) or *JSP* (almost exclusively in *Java*), there are many systems that are still implemented in *PHP* – one of the best examples being *Wikipedia* or *Facebook*. Additionally many in-house intranet systems are using *PHP*.

Obvious problem of commercial systems implemented in *PHP* is performance and integration with modern technologies. *Phalanger* is an ideal solution to both – it analyses and compiles *PHP* programs and enables them to easily integrate with *.NET* applications and libraries. Moreover, *Phalanger* is designed in the way it can be compatible with whole *PHP* semantics, which is not true for *Facebook's HPHP* [**7**] that sacrifices compatibility for simple compilation process and performance.

For the reasons stated above we concluded that *Phalanger* is good choice for research and our goal of improving its compilation process would benefit both scientific and business applications of the platform.

The main goal of this thesis is to design static program analyses for *Phalanger*, which would help the compiler to optimize resulting programs. Especially it should refine static analyses that are already performed by *Phalanger* and devise new analyses, which were not implemented in the project before. For instance, *control-flow analysis* would help to solve flow-sensitive problems and *type analysis* would help to remove most of dynamic operations that would be otherwise performed in runtime. The concept should be tested on experimental implementation, which would include a representative subset of PHP's features.

The rest of this thesis is organized into chapters as follows:

- In Chapter 1 we familiarize the reader with general overview of topics that will be used frequently in the rest of the work or reader's basic knowledge of which will be presumed throughout the work, for example *PHP* language, *CLR*, *CIL* and related work.
- In Chapter 2 we present performance analysis of *Phalanger* using synthetic benchmarks that detail differences between *PHP* implementations feature by feature.
- In Chapter 3 we briefly describe *control flow analysis*, which we will need for *type analysis*. This analysis is not currently implemented by *Phalanger*. We point out key problems of performing the on *PHP* programs and where needed presenting solution to these problems.
- In Chapter 4 we present our approach on type analysis of *PHP* programs built on top of control flow analysis. We first show common situations that type inference has to deal with in order to be efficient and then describe the formalization behind the algorithm in detail.
- In Chapter 5 we introduce our prototype implementation that acted as a testing platform. We describe implementation of type analysis algorithm.
- In Chapter 6 we present our result on *AST*-based compiler design that prioritizes modularization, extensibility and parallelism. This chapter also briefly presents the architecture of the prototype.
- In Chapter 7 we perform performance benchmarking of the prototype implementation, showing that simple programs can be efficiently compiled.

1 Background and related work

This thesis focuses on analysis of PHP language, compilation of *PHP* programs and design of *Phalanger* compiler. Then, it moves to translation of results of this analysis into efficient algorithms that perform programmatic static analysis of *PHP* programs and subsequent optimizations, which improve performance of the resulting *CIL* object code. The main body of the work expects the reader to have at least a basic knowledge of several topics that are discussed in this chapter.

In the beginning, we were not considering to include some of sections that are present herein and rather to reference other sources. However, we found out that there is no proper literature on the subject or, at least, there is none with a point of view we were finding appropriate for coverage of this thesis and which we were trying to attain.

Therefore, this chapter familiarizes the reader with dynamic languages and their dynamic operations and type systems. Since the dynamic language semantics are designed without strict differentiation between runtime and compile time, we discuss main problems the dynamic language compilers are facing.

Second, we introduce *PHP* language, which is ubiquitous in the entire work, detailing the language features and constructs that are being of concern of static analysis and optimization presented later. *PHP* has several very unique features that are interesting or difficult for compilation and static analysis, e.g. the type system or call context semantics.

Afterwards an introduction of *Phalanger* follows. It is a fairly large project, which encompasses many different technologies and types of software from *PHP* extension hosting, through language compiler, to extensive implementation of *PHP* class library.

Finally, we describe projects similar to *Phalanger* and research works that have close relationship with this thesis. This includes especially *HPHP* [**7**] implementation of a *PHP* compiler, which is developed and used in production environment by Facebook Inc. Another project, *PHC* [**15**], is research project that is targeted on static analysis of PHP code without changing language's semantics, which similar to our goal.

1.1 Dynamic Languages

Notion of dynamic languages probably first emerged with *LISP* language in 1958 when McCarthy started his work, which eventually ended in publishing his paper [**16**]. Key feature of *LISP* was its *EVAL* function, which allowed forms to be evaluated and executed dynamically at runtime. *EVAL* function continues to be a typical feature of dynamic languages until today.

The word dynamic in the name of this class of programming languages denote a presence of operations and features, that are normally considered to be performed by language's compiler, are postponed to runtime and are performed using the actual state of program's execution. These features can comprise of a combination of adding new code to extend the program, extending definitions or by modifying the type system. This definition is vague since almost none of these features are present in all of the languages considered dynamic and many features are also present in static languages.

1.1.1 Dynamic Type Systems

Most of dynamic languages use dynamic type systems, in which the majority of type-related operations is deferred to runtime in the language's semantics[1] – in other words every value carries its type and possible other information with it. This enables the language and it's runtime to:

- Use of untyped variables, arguments and object properties.
- Make operators and function calls late bound.
- Change type definitions at runtime as a language feature.

Omission of the type information leads to a more succinct code and faster application development. A programmer then produces only an operational logic – structure and order of control statements, functions and operators along with a data flow. Interpretation of the program is then made at runtime when operators and functions are late bound based on the execution state. The downside is, that it makes the program code less clear and understandable and makes programmer more prone to making errors as the value typing acts as a form of specification testing.

Late binding is known from static languages in form of virtual methods, in which case the method's owner and its interface are known by the caller and only the implementation is chosen at runtime. However, in a dynamic language late binding is omnipresent – variables, functions, methods and properties can be bound to their implementation at runtime. In case of objects, so-called duck typing is often used. This is a form of dynamic typing in which object's set of methods and properties determines valid semantics, rather than checking type's entire structure as in structural typing or type's name as done in nominative typing.

Variable function and type definitions again enable the programmer to produce a less structured code with a lower level of abstraction. For instance, when program's behavior need to change with the configuration we can select a function definition out of several choices and then declare it, avoiding the need to create switch statements. In this way an existing program can be easily altered without need to change existing functions. This can be regarded as inherent polymorphism. In contrast, a static language program needs to explicitly implement a level of abstraction (such as interface or base class and several derived classes).

1.1.2 Dynamic Operations

Specific property of dynamic languages is presence of syntax for dynamic operations. Such operations are not present in static languages at all or are supported programmatically only by their class libraries. Most common dynamic operations include:

- *EVAL*, a function that evaluates a string as it was an expression of the programming language.
- *Indirect call*, which calls a function based on a variable representing its name.
- *Indirect variable access*, which accesses a variable based on its name.

As mentioned earlier, *EVAL* was first used in *LISP* and it did not operate on generic strings but rather on other parts (forms) of a program, which were syntactically correct. It was later implemented by compilers. In general case, such as in *Perl* or *PHP*, *EVAL* can take any string that is supposed to be syntactically correct, thus it is not guaranteed for the executed code to be inferable at compile-time.

[1] Many such operations can be still performed by language's compiler after the code is properly analyzed – see Chapter 3 for further details. We emphasize this, because languages are designed with this assumption.

This notion of *EVAL* inherently causes problems for compiler's static analysis as it cannot determine what an *EVAL* can possibly do since the *EVALed* code has similar semantics as the rest of the method's code. Thus, an *EVAL* operation can cause almost any semantically observable side-effect including, but not limited to, following operations:

- Changing the caller's stack frame - local variables and arguments. This also includes creating a new variable and unsetting one in dynamic function scopes.
- Changing a value of a global variable to an arbitrary value. Again, includes creating and unsetting a global variable.
- Calling a global function, including built-in ones, with arbitrary arguments. This can eventually produce any side-effect caused by all known functions.
- In some languages declaring of a new function, changing set of valid declarations. This also affects sets of possible side-effects of other dynamic operations.

An important property of *EVAL* is that if it is used in a subroutine it makes the subroutine behave like *EVAL* for arbitrary values of arguments. This can only be negated by further analysis, but if we cannot infer the exact code the *EVAL* will run with, which would make possible to remove the *EVAL* completely, we cannot make any statement on how the subroutine will affect program's state. In that case, we can only state what the subroutine will do before it executes an *EVAL* and what it will do after it executes an *EVAL*. [2] Inherently, this property is valid for all callers of the subroutine all the way to program's entry point.

Indirect call is a dynamic operation that makes use of late binding and calls the function using identifier stored in a variable rather than by constant identifier, thus indirectly. Side-effects possibly caused by indirect call are those caused by functions that are currently defined. Moreover, variable where the function identifier is stored is known and so are the arguments of the call. Therefore, indirect call is only a small subset of *EVAL*'s capabilities and set of side-effects is more evident to the compiler. However, if any function callable by the indirect call contains *EVAL* operations in its call tree, the indirect call operation inherits this property for arbitrary function identifier.

Indirect variable access is similarly access to a local or a global variable using an identifier stored in a variable to a value specified on the right hand side of the assign statement. Again, this is only a fractional subset of capabilities of *EVAL* statement, since both variable identifier and new value are known to the compiler. Contrary to indirect call, indirect variable access cannot itself execute *EVAL* and does not inherit *EVAL*'s properties when the *EVAL* is present in the program.

1.1.3 Summary

In overall, dynamic languages can improve development effectiveness by producing more succinct and inherently polymorphic code, which is much easier to change than code written in a static language. This is mainly caused by absence of variable typing, absence of declaration flow known from languages such as *C/C++*, possibility to create conditional declarations that are evaluated at runtime.

On the other hand, such code is less understandable, more prone to containing errors and generally harder to debug. Using the rule of a thumb, dynamic languages are more suited for rapid

[2] This fact forms a basis of our method of dealing with *EVAL*, which is discussed in section 4.2.5.

development of small projects with unclear or frequently changing specification, implemented by individuals or very small teams of developers.

1.2 PHP language

PHP [**1**] is currently one of the most popular programming languages used primarily for web application development. It includes a vast set of libraries which provide support for almost every technology which is relevant to web development. This, along with easy-to-learn syntax and semantics, can be attributed to *PHP's* global success. For example, in [**17**] it is claimed that *PHP* serves web pages for tens of millions of domains.

In the rest of this section we will provide a brief description of *PHP* and its syntax and semantics. It is important to note that *PHP* does not have any standardized specification – the official implementation is the only specification available. This fact is even more amplified by tens of independent developers working on the official implementation. Because of this, we will try to provide a brief description of the language's overall design, syntax and semantics, including our knowledge of internal working of the language. We will later refer to this section when we will be analyzing *Phalanger* compilation process and describing static analyses.

1.2.1 Language design

Syntax of PHP language is mostly a hybrid between *C* and *Perl* with some construct coming from *C++* and *Java* and a few being *PHP*-specific. The basic structure of the source code is divided into source files that are called scripts, each of which being alternating sequence of *HTML* code blocks[3] and *PHP* code blocks. In Snippet 1 we show the basic structure of *PHP* program with code embedded into *HTML*.

```
1:      <html>
2:              <head><title> <?php echo $title; ?> </title></head>
3:              <body>
4:                      <?php
5:                              for($i=0; $i<10; $i++) { ?>
6:                                      <p> <?php echo $i; ?> </p>
7:                                      <br />
8:                              <?php } ?>
9:              </body>
10:     </html>
```

Snippet 1 - Basic structure of a PHP program using embedded HTML.

PHP scripts are usually described as consisting of embedded *PHP* code into *HTML* and they even look as such. However in order to describe syntax, we have to treat *HTML* blocks as merely special syntax for *Echo* statement. This is internal working of the code and since embedded blocks can be inserted into any place where a statement can be, it is a statement. On lines 2, 4-5, 6 and 7 are *PHP* code blocks. All the text in the beginning of the script, between the *PHP* blocks and in the end of the script is treated as echo statements with the respective text as an argument. This viewpoint is important because *HTML* blocks can occur in all places where a statement can be – in loops, functions and even

[3] This the most frequest use. Other use of these blocks is implementation of text-based protocols such as SMTP or FTP.

methods. Bearing this in mind, we can divide a *PHP* script (also called global code) into a sequence of global statements, which are divided into:

- Declarations – global function declaration and class, interface or trait declaration.
- Control statements – *if* statement, *switch* statement, *for* loop, *while* loop, *foreach* loop and *goto* statement etc.
- Assign statements – direct or indirect assign to a variable or object property.
- Call statements – direct or indirect call of global function or object method.
- *Eval* statement – which evaluates an argument as if it was *PHP* code.
- *Echo* statement – which outputs given string to standard output. This also includes the embedded HTML blocks.
- Include statements – both strict (*require*) and non-strict (*include*) along with onetime variants (*include_once* and *require_once*).
- Several other statements which we will not need for purpose of this work (exception handling).

1.2.2 Type System Basics

PHP has type system that strictly differentiates between primitive types and objects. This means that primitive types are not considered as objects and have no object-like methods or properties in contrast to some object-oriented languages. *PHP* uses following types:

- *Boolean* – a simple binary value, internally represented by 32-bit or 64-bit value.
- *Integer* – a signed integral value represented by 32-bit or 64-bit value depending on environment for which PHP was compiled.
- *Float* – a floating point value conforming IEEE 754 double precision format.
- *String* – a string of bytes (sometimes called binary string) with attached length.
- *Array* – a map of *String* or *Integer* keys to arbitrary values. It is used as multi-purpose data structure.
- Object – a complex data type consisting of set of methods and properties that are initialized from object's inheritance. Objects in *PHP* are more corresponding Java's objects than those of some other dynamic languages.
- *Resource* – a set of managed types that hold external resources.
- *NULL* – empty value.

Value semantics correspond with internal implementation, so we will describe it in brevity and without mentioning details we won't need. Each value is represented by a *C* structure, which contains integer type of the value, union of all possible types of values and reference counter. These values are shared between variables (global variables, local variables and object's properties) using the reference counter. This is possible in interpreter environment and needed because arrays are passed by value in language semantics with implementation using notion of copy-on-write values to improve performance.

```
1:      $a = 1;
2:      $b = &a;                        // $a shares value with $b
3:      $c = array(&$a, &$b);   // both values in $c share value with $a
4:      $x->prop = &$a;         // $x->prop shares value with $a
5:      $c[1] = 2;                      // all variables change to 2
```

Snippet 2 - References in PHP.

Reference semantics are very specific for *PHP* – Snippet 2 shows that a value can be shared between multiple types of variables – local variable (or global variable), array values or object properties. This is true for all types. Moreover, as line 5 shows, there is no syntactical difference between using dereferenced and non-referenced values in many cases. Some operations, as function calls, automatically dereference the values, but a programmer has to create references carefully because it may lead to very awkward behavior of programs.

Objects in *PHP* are typed by a system of classes with single inheritance model and interfaces similar to *CLR* or *JVM* languages. In such system, each class inherits from single parent class and multiple interfaces. Interfaces do not contain any fields and constructors, they only declare set of methods with identifiers and signature (and possibly interface constant discussed later).

This system makes object-oriented programs much more readable and understandable than multiple inheritances known especially from *C++*, where a class can inherit from multiple other classes. In order to enforce encapsulation in general case, such models allow including data and functions separately for each parent class, which can lead in an extreme case to multiple instances of a base class being contained within a single object. This causes severe ambiguity and modern languages tend not to include this behavior, since it is memory inefficient and does not cope well with garbage collection.

PHP includes type-checking using duck typing as do other dynamic languages. Such type check is performed at runtime when an object member is about to be accessed. In contrast to static languages, duck typed languages check only the member's name and therefore any type containing the member are valid for the operation. It is then questionable whether interfaces are needed in *PHP* since duck typing provides a form of proto-interfacing. In our opinion it was merely a design decision as developers are familiar with interfaces from other languages. Thus, interfaces are just a syntactic and semantic sugar that could be implemented easily using only duck typing.

1.2.3 Global functions

Global function declaration is a statement in *PHP* since there is no special place for declarations as in typical static languages. Outcome of the statement is declaration of function of the specified name, which will be valid at runtime from now on. The statement cannot replace existing function and an error is raised if the program is attempting to do so. Function declaration is a valid statement even in other functions; however the function's scope is always global.

```
1:      function foo($a, $b:MyClass) {
2:              return $a + $b->property;
3:      }
```

Snippet 3 - Simple function declaration.

Snippet 3 shows a simple function declaration introduced by a keyword function, which followed by function's identifier. Function arguments have the same *$* prefix as variables do have and are not typed. Type constraint can be specified, limiting the argument to a certain class. This is evaluated at runtime each time the function is executed and is not generally used by developers. The function's code (block of statements) is enclosed in mandatory braces.

Signature of a function is only the function's identifier and does not include argument count and types in the way many other high-level languages do. Number and naming of a function's argument has meaning only for the function itself, because a function can be called with arbitrary number of arguments without the names specified. The names however affect set of initial variables of the function's scope.

1.2.4 Classes

In *PHP*, similarly to other dynamic language, objects have special methods which are invoked each time a dynamic operation is about to happen – so called magic methods. This makes seemingly static *PHP* classes truly dynamic. For instance, magic methods can handle call to an undefined method or property, call of the object as if it were a function or object cloning.

```
1:      class MyClass
2:      {
3:              private $a;
4:
5:              public function foo($x) {
6:                      return new MyClass($this->a + $x);
7:              }
8:
9:              function __construct($a) {
10:                     $this->a = $a;
11:             }
12:     }
```

Snippet 4 - Simple class declaration.

Simple declaration of a class in *PHP* is seen in Snippet 4. This declaration contains single property (declared using a visibility keyword private on line 3) *$a*, a method foo (lines 5-7) and constructor with single argument (lines 9-11).

Methods in *PHP* are inherently virtual and a function automatically overrides the same function defined in base class, interface or trait (see further). Set of object's methods is constant throughout its lifetime – a method cannot be added, removed or have its definition altered. For instance *__call* magic method can handle calls to unknown functions.

Set of properties is variable – new public properties can be added to the object regardless of being declared. Already declared properties will be written to if they are accessible. Such dynamic properties are called auto-properties and can again be handled by *__set* and *__get* magic methods.

Constructors (magic method with special name *__construct*) are somewhat new to *PHP*, before their introduction they were simply methods which had to be manually called after object's creation. *PHP* also supports destructors (special name *__destruct*). Methods are declared using function keywords

and are similar to global functions. The only difference is a presence of a special $this variable which is used to access fields and methods of the current instance.

New object are created using new operator (line 6), similarly to C# or Java, with class identifier following. This class identifier is late bound and need not to be known when the program is being parsed. There is also an indirect equivalent of the new operator. As mentioned above, prior to PHP 5, the new operator only initialized the object with default values and did not invoke any constructor.

```
1:      class MyOtherClass extends MyClass {
2:              private var $b;
3:
4:              function __construct() {
5:                      parent::__construct(0);
6:                      $this->b = 1;
7:              }
8:      }
```

Snippet 5 - Class inheritance.

In Snippet 5 we demonstrate simple class inheritance, which is similar to other languages. The only difference is that base class constructor execution is not mandatory and additionally can happen everywhere in the constructor. Also, constructor can be invoked by other functions. When a base class is not specified, it is assumed that the class is inherited from the common base class called *stdClass*. *StdClass* class is important, since it is used for creating anonymous objects which are changed using auto-property functionality and used internally by algorithms of even globally.

Static members

PHP classes can declare static members, that aren't dependent on actual instance of the class. The specific feature of *PHP's* implementation is that instance methods and static methods can be used interchangeably – an instance method can be called in static manner and vice versa.

1.2.5 Interfaces and Traits

Interfaces allow specifying which methods must a class implement to be used in a way represented by the interface. It consists only of set of public methods without specified implementation, which allows implementing more than one interface by a class without encountering the same paradox as with multiple inheritance we have mentioned above. Additionally an interface in *PHP* can declare so-called interface constants, which are similar to class constant except that they cannot be overridden in a derived class.

```
1:      interface ISomethingWriter extends ISomethingDoer {
2:              const c = "MyValue";
3:              public function WriteSomething($text);
4:      }
5:
6:      class DoerAndWriter implements ISomethingWrite, ISomethingDoer {
7:              public function DoSomething() { … }
8:              public function WriteSomething($text) { … }
9:      }
```

Snippet 6 - Interface declaration

Declaration of an interface is similar to other languages as seen in Snippet 6. Lines 1-4 show an interface with a class constant and single method extending other existing interface. Lines 6-9 show class implementing two interfaces. Syntax is similar to Java and other languages.

Traits are being introduced in PHP 5.4 which is at alpha stage in time of writing of this thesis and therefore we won't be discussing them in detail. Notion of traits originated in *Self* programming language [**18**] and are, similarly to interfaces, a collection of methods. In contrast to interfaces, traits also contain implementation of those methods. This allows better code reuse without breaking single inheritance. However, PHP's implementation of traits is essentially a code template that is inserted to the using class with semantics being checked in its context at runtime, thus making another completely dynamic feature.

1.2.6 Inclusion Statement

Scripts are unit of modularization of *PHP* programs and it syntactically and semantically resembles header inclusion in *C/C++* language preprocessor. However, inclusion statement is done at runtime and is capable of using caller's variable scope, be it other script (thus using global scope) or a function or a method (using the local scope). Since inclusion statement is evaluated each time the program is run, it is more related to *EVAL* than to static inclusion.

1.3 Common Language Infrastructure

Common Language Infrastructure [**14**] is a standardized runtime environment for programming languages. Introduced by *Microsoft* as its proprietary *.NET Framework* [**19**], it is now implemented as by open-source project called *Mono* [**20**]. *CLI* primarily consists *of Common Intermediate Language* (*CIL*), which is an object-oriented assembly language operating on virtual stack-based machine, *Common Language Runtime* (*CLR*), which compile the *CIL* object code into machine code using a *JIT* compiler, and *Base Class Libraries* (*BCL*), which is vast class library shared by all *CLI* languages.

Main languages that are implemented on *CLI* stack are *C#*, *C++/CLI* and *Visual Basic*, but there are many more languages such as *F#*, *IronPython*, *IronRuby* etc. All languages share the foundations in *CLI* and are therefore inherently interoperable.

1.3.1 Compilation Process

Common Intermediate Language is fully-featured assembly-style programming language that natively supports all programming constructs supported by *CLI*. It supports declaration of classes, methods and properties, interoperability using *P/Invoke*, unmanaged memory access and exception handling. It does not support higher programming language constructs, such as loops. *CIL* programs are compiled into *CIL* object code, a binary representation that is also output of other programming languages. This object code forms an executable or a library and is transformed into machine-specific code during runtime or installation.

CIL instructions work with a stack-based virtual machine, which relieves the guest language compiler of many operation that would need to be done in ordinary machine-based assembly, especially low-level optimization such as inline expansion, register allocation, instruction prefetching etc. This virtual machine has conscience of types and during the transformation into machine code using just-in-time (*JIT*) compiler, types on evaluation stack are checked.

Emission of *CIL* programs by compilers is simplified by integrated set of classes that allow emission of *CIL* programmatically, without the need to worry about the exact binary representations.

1.3.2 Types

There are two main groups of types defined in CIL. Reference objects are passed by reference, which are managed by a garbage collector, i.e. are located freely in managed memory and have lifetime unrelated to currently executed procedure but rather are kept alive until someone has reference to them. Second group are value types, which are not managed but are located directly on the call stack, in objects or in arrays. Value types are inherently not garbage-collected because their lifetime is always bound to something else.

However, value types are much faster than reference types, because of the lack of indirect access and the lack of associated complexity of garbage collector. In memory representation of value types does not include type information as reference types have. However, value types can be wrapped into a typed envelope, a process called boxing and opposite called unboxing. Therefore, all .NET types can be treated as an „*object*“, a base class of all values.

1.4 Phalanger

Current publicly available version of *Phalanger* traces its origins in a software project called *PHP.NET* made by students of *Faculty of Mathematics and Physics* of *Charles University* in Prague, Czech Republic, in 2004. In its preliminary version, the project was intended to be more an advanced interpreter of *PHP* than compiler since it parsed the source code on demand and CIL object code was used only for caching of already parsed source. However, *CIL* emission support in *.NET* is not intended for such usage as it performs very intensive correctness analysis, which can be very time-consuming. Because of this the idea was quickly scraped and real compiler was implemented.

The final version of the project was focused on compilation, reimplementation of PHP language on *.NET* along with its class libraries, native extension support and integration with existing technologies. It was intended to run on *.NET* runtime version 1.1. After the university project ended, Tomáš Matoušek and Ladislav Prošek continued their effort and updated the project to *.NET* 2.0 runtime and eventually published the source code on Codeplex. The project was renamed to Phalanger because of PHP's license obligations.

After some time of unsuccessful efforts to promote the project commercially, the authors were recruited by Microsoft's *.NET* development division. The project was taken over in 2007 by another group of students of the same faculty, who form the core development team until today, the author of this thesis being one of them.

1.4.1 Basic architecture

Phalanger, same as its *PHP.NET* predecessor, is formed by a system of several different parts, which encompass several fields of computer science. Main parts of *Phalanger* are:

- **Compiler.** The compiler translates *PHP* source code into *CIL* object code. Despite dynamic semantic compilation this is not always possible, the compiler manages to transform most of the code into a form that uses Phalanger's dynamic runtime. The rest is left in textual PHP form and is compiled on-demand at runtime.
- **Runtime.** Since the dynamic languages were not primarily intended to be part of .NET Framework, the CLI does not support directly most of dynamic operations that are performed by dynamic languages. Because of this, Phalanger implements fairly large system of functions that allow the compiled code to perform such operations without interpreting

the code. This includes indirect operations, E*V*AL operations, script inclusions, operators, dynamic dispatch etc.

- **Class Library.** Source code of *PHP's* class library, implemented in *C*, could not be reused and had to be implemented from scratch. The reimplementation brought further benefits such as type-safe assemblies and usage of PHP functions from *.NET*. Moreover *.NET* attributes and typed library functions allow the compiler to perform optimizations that are not possible on original implementation.
- **Native Extension Support.** PHP comes with vast array of libraries, called extensions, which are developed by PHP team or third parties and there are even proprietary extension. Reimplementation of all extensions is thus virtually impossible and so Phalanger includes an interoperability layer, which allows using compiled *DLL's*. This layer takes advantage of *C++/CLI* and translates the internal calls that *PHP* extensions make to *PHP* core into Phalanger calls and vice versa.

There are other tools and programs that we won't be mentioning and that are designed to support developers working with Phalanger.

1.4.2 Compiler

Phalanger compiler uses *Gardens Point Parser Generator* [**21**] and *Gardens Point Scanner Generator* [**22**] which allow using *C#* as a main language in which is the compiler implemented. Thanks to the nature of *CIL* and the language as a whole, *Phalanger* does not use any other representation than *Abstract Syntax Tree* (*AST*). Main reason for lack of need for low-level linear representations is the fact, that low-level optimizations are performed by *JIT* when compiling *CIL* programs from object form into machine code. For more information about CIL emission the reader should refer to [**23**] [**24**].

AST architecture in *Phalanger* recognized three main types of nodes – *Global Code*, *Statements* and *Expressions*. Global Code represents a code in single script file and is root node of the *AST*. Each expression is capable inferring its type and if possible, evaluating its value.

The only notable static analysis that Phalanger compiler performs is so-called script inclusion analysis that removes some dynamic sites of functions.

1.4.3 Script Inclusion Analysis

Script inclusion statement is basis of modularization of PHP programs. To summarize its description from section 1.2.6, script inclusion statement executes the target script in the current context, being it global code, global function of object method, at runtime. This statement inherently changes program's state – set of included scripts, values of global and local (if applicable) variables and set of valid declarations. The latter set is monotonic throughout the program's execution (see section 1.2.3) so that only possible change is introducing of a new declaration.

In order to infer side-effects at compile-time, a compiler has to analyze side-effect of each statement inside the included script, including other inclusion statements. This is recursive problem which requires analysis of program's flow and is hard to solve by static analysis. Phalanger chooses less generic approach, since it is inspecting only one class of side-effects – function and class declarations – and treats the script file as one unit, eliminating need for classic flow-based analysis. The notion of script file as sequence of statements then contracts to a tuple of declared functions and included script files.

The goal of this specialized analysis is to infer a proof that a dynamic site references particular declaration, thus making it possible to make the operation early bound. It is inherently limited to taking into account only unconditional global inclusion statements and unconditional declarations. In

Algorithm description

To provide insight on how the algorithm works, let us consider following example of resolving dynamic site D targeting declaration identified by identifier N (which is tuple of textual name and class of declaration) in script file F, which requires following steps:

1) Enumerate set of preceding inclusion statements I_0.
2) Enumerate set of preceding declarations E.
3) Set $i = 0$ and while Ii is not equal to I_{i+1} do:
    a. Set I_{i+1} as union of I_i and set of known target scripts of unconditional inclusion statements contained in I_i.
    b. $i := i + 1$
4) $I' := I_i$
5) Set E' as union of E and set of unconditional declarations in I'.
6) If N is in E' as e, bind D with e.

Properties

The script inclusion analysis is a form of control-flow analysis that does not recognize branching nodes but is capable of dealing with different control flows for different script files and therefore is able to infer global unconditional properties of the analyzed source code, such as declarations that are guaranteed to be made in specific point of the program.

In many cases it produces satisfactory results, but as programs grow larger, attaining results is progressively harder for this algorithm. This is mainly because of the presumption that call dependencies (i.e. the fact that a function is called by specific dynamic site) often go forward, i.e. that function is used after it is declared. This is not valid in many real world programs, causing many call sites to be left dynamic.

Moreover, the analysis deals only with static structures – functions and static methods. Many PHP programs use object-oriented approach and these programs mostly do not benefit of this analysis and are left with low performance of dynamic dispatch (which is slow especially for instance methods).

1.4.4 Runtime

Phalanger is not an interpreter and therefore does not execute the code completely dynamically. Control statements of the language are always compiled into *CIL* and operations that cannot be succinctly expressed in *CIL* are done using Phalanger runtime, as set of functions and objects that emulate behavior of *PHP* interpreter. Most frequent operations performed by this runtime are dynamically dispatched functions and methods, operators and type conversions.

In [**25**] an optimization of the runtime using inline caching, which is implemented by DLR technology [**26**] was introduced, which would eventually reduce complexity of dynamic operations performed. This was implemented in Phalanger during writing of this thesis and improved some of the worst performing operations in Phalanger runtime.

1.4.5 Interoperability

Phalanger, apart from being faster than original PHP implementation provides notable benefit with introducing two-way interoperability with CIL languages. Programs compiled by Phalanger can be used from C# and other languages both directly using internal Phalanger objects and representations and using one of supported interoperability layers. *CIL* programs can be used from PHP programs using *PHP/CLR* syntactical and semantic extensions to the original language.

Duck-type interoperability [**27**] enables the static language users to declare strongly and statically typed interfaces which wrap *PHP* objects or scripts. The runtime uses the type information provided to convert native *PHP* values into representations common in other languages. The *DynamicMetaObject* interoperability [**28**] takes advantage of new *dynamic* keyword in *C#* and other languages, which allows to dynamically call *DLR* objects using the same syntax as static objects are used.

PHP/CLR language extension provides natural access to CIL objects that are passed to PHP code and also other .NET values. These are then treated as if they were PHP objects and even can have dynamic properties attached to them.

1.5 Other PHP compilers

Phalanger was the first fully featured compiler of PHP. However, since PHP language is very popular, several other compilers were created. We will briefly describe these technologies and we will not discuss their performance or internal implementation, since we will mention that during performance analysis present in Chapter 2.

1.5.1 Roadsend

Roadsend compiler [**29**] is a reimplementation of *PHP* language, similar to what *Phalanger* is, with the difference that it is targeted to native code and is therefore incapable of dealing with some *PHP* features. Moreover it is targeted to be compatible with PHP 4 and it has better performance than it, but when compared to newer versions of *PHP*, it is clearly lacking performance and compatibility.

The development team is currently preparing a new version, which would include several static analyses taken from *PHC* and *LVVM JIT* compiler for supporting dynamic featured. However, it was unfinished and not capable of producing working executable files for the whole time of writing of this thesis.

1.5.2 HPHP

Globally known social network Facebook, operated by *Facebook Inc.*, is partially implemented in *PHP* and the company started to improve performance of their website by implementing *PHP* compiler called *HipHop for PHP* [**7**]. This compiler transforms the low-level representation of the language into C++ code, which it then compiles using a standard compiler. This inherently pushed creators to limiting some dynamic features of *PHP* such as removal of *EVAL*, dynamic inclusion and library functions that have access into caller's stack frame. It also alters some of language's special semantic features such as integer values that overflow becoming double and dynamic object properties.

HPHP performs very aggressive optimizations. Most important is type inference that is able to remove dynamic dispatch in most cases. If the type analysis is not possible, *HPHP* has fairly fast support for dynamic dispatch. According to authors of the project, processor performance costs

dropped by 30% compared to the same code run by PHP, which is a notable improvement for big web service provider.

1.5.3 PHC

PHC [**15**], an open source *PHP* compiler is an experimental compiler of *PHP*, which again produces *C* code that is then compiled by a standard compiler. It is build, as *HPHP* does, on top of *PHP's* source base, taking place of the interpreter, compiling the PHP byte code into a C source file.

More interesting is its data-flow analysis branch presented in [**8**], which targets to optimize the program in more static manner. However, this branch does not support many language features and is still experimental.

1.6 Dynamic Languages for CIL

Phalanger is not the only dynamic language that was implemented for *CIL*. The complication that hinders implementation of dynamic languages is inherent *CLR's* orientation to statically typed languages. This is partially solved by introduction *of Dynamic Language Runtime* (*DLR*) [**26**] by *Microsoft*, which provide efficient implementation for dynamic features that are often needed by dynamic languages. Most notable language implementations using it are *IronPython* [**4**] and *IronRuby* [**5**], both created by *Microsoft* and currently supported by community.

1.6.1 DLR

The motivation behind *DLR* is to enable dynamic programming languages to be easily implemented on top *.NET* (and therefore *CIL*). This task is very complicated, since implementing a sound compiler of dynamic language, or even a fast interpreter can be very time-consuming. Therefore, DLR provides a complete back-end and intermediate representation for a compiler.

Typically, a compiler that uses *DLR* parsers the source code and performs semantic analysis and then instead of emitting *CIL* as typical *CLI* compilers do, it transforms the *AST* into DLR's own syntax tree. This syntax tree can be both interpreted and compiled by the DLR when required. Dynamic operations, such as dynamic dispatch, are implemented efficiently using polymorphic inline caching that is able to cache the binding based on argument types.

1.6.2 IronPython

IronPython, a reimplementation of Python [**30**] language, was the original language for which the *DLR* was created. It existed before the DLR, having a similar operational stack as Phalanger currently does. Then, the compiler was changed to support *DLR* for several times as DLR had several incompatible versions (it was in fact part of *IronPython* code base in the beginning.

IronPython most notably sports a so-called light compiler, which is *DLR*-enabled interpreter capable of quick interpretation of code that was not yet compiler – typically code that is executed using *EVAL* operation. Similarly to *Phalanger*, *IronPython* allows *Python* programs to use whole set of *BCL* and conversely

1.6.3 IronRuby

IronRuby is an implementation of Ruby language, which is mostly known for *Ruby on Rails*, a *MVC* framework for Web applications developments. It supports *Ruby's* whole dynamic type system, but does not support advanced features, such as continuations, which are not possible to efficiently implement in *CIL*.

2 Performance Analysis

This chapter targets to identify types of programs, language features and operations that do not perform well in the version of *Phalanger* that was taken as referential – published on *Codeplex* as Change Set 86245 from March 17 2011. Results of this analysis will allow us to propose methods for optimization of *Phalanger's* performance and proposal of static program analyses that would allow it. This chapter described the actual process of problem analysis we have used.

We analyze performance of compiled programs as a black box by using both synthetic benchmarks and compare them with results of other PHP runtime environments. Then, knowing performance of language features, we analyze source code of real-world programs implemented in PHP. This gives us an estimate of real distribution of operations that we have measured in synthetic benchmarks.

Then, we describe the most important operations that need to be optimized both at runtime and by performing static program analyses. We will use these conclusions in following two chapters that describe static analyses in general and type analysis of various elements of the language.

2.1 Synthetic Benchmarks

Synthetic benchmarks (called *Micro Benchmarks* in *Phalanger's* testing suite) are synthetic tests, that are implemented to test common operations and features that PHP has and that we are interested in. Each benchmark consists of a small test that is iterated in a for-cycle.

We run each benchmark three times and measure complete execution time including the loop. In the end we take minimum value for each benchmark – we presume that time measurement is reasonably precise and the major sources of error are background tasks in the operating system. For this kind of error cause minimum value is a rational heuristic.

Additionally we subtract results of empty loop benchmark from results of other benchmarks in order to remove a bias attributed to result of this test and produce more accurate results for the specific operation.

Tested Technologies

Technologies we have tested are the following:

- *PHP* 5.3 on *Windows* with *WinCache* enabled
- *HPHP* (*HipHop* for *PHP*) on *Linux*
- *Roadsend* (original) on *Windows*
- *PHC* on *Linux*
- *Phalanger* v2.1 on *Windows* and *.NET*

We did not include original PHP without any cache extension since its runtime performance is similar and it adds a strong bias with executing compilation of whole source code during each execution. Additionally we did not test Phalanger on Linux since the version we have been testing did not fully support Mono runtime environment.

In case of *PHP* we did not differentiate between Windows and Linux version when running micro benchmarks since the differences would be little and our goal is not to measure exact performance, but to analyze differences and to narrow possible targets for optimization.

Testing Environment

During the benchmarking we have used single configuration for all tested technologies and environments in order to avoid a bias in results. Because of technical reasons, we did not install *Linux* operating system on the physical machine and rather used virtualization software with hardware virtualization support for both Linux and Windows environments. Since virtualization does introduce a small penalty to some operations and larger to others, the results may not be near the possible real machine results. However in our opinion this penalty of virtualization is rather small and the differences between operation penalties are irrelevant for testing *PHP* implementations.

Configuration of the physical machine was as follows:

- *Intel Core i7 Q820 @ 1.73 GHz* (4 cores, 8 threads)
- *16 GB of RAM @ 1333 MHz* (dual channel)
- *Intel 320s SSD* Drive (300 GB)
- *Windows 7* (64-bit)
- *VmWare Workstation 8*

Configuration of the *Windows* virtual machine:

- 2 virtual processors (2 threads)
- 2 GB of RAM
- 40 GB virtual HDD (located on physical SSD)
- *Windows Server 2008 R2 Enterprise* (64-bit)

Configuration of *Linux* virtual machine:

- 2 virtual processors (2 threads)
- 2 GB of RAM
- 20 GB virtual HDD (located on physical SSD)
- *Ubuntu 10.10 Desktop* (64-bit)

Notes on Benchmark Results

While testing, we encountered two problems – one with *PHC* and one with *Roadsend*. *PHC*, although it is very advanced compiler that perform advanced static analyses, is not finished. We weren't able to run micro benchmarks with optimizations using *PHC* because of several unsupported language features that are essential to these benchmarks. Consequently, we weren't able to run either application benchmarks Therefore we did not include any results since with optimizations disabled, PHC uses original PHP runtime to interpret the code.

In case of *Roadsend* the problem is that the implementation is very old and does not support several *PHP 5* language features, which caused application benchmarks not to work along with several micro benchmarks. Additionally, it is implemented in *Bigloo Scheme*, which could be source of decreased performance even if the compiler itself is capable. Although the authors are working on new version, that targets to be very advanced compiler with garbage collection and just-in-time compilation, it is

in development for several years now and was not able to run any benchmarks since of incomplete code generation.

2.1.1 Loops

Apart from being one of the basic programming constructs, we need to explicitly measure loop performance to remove the loop overhead from other results to clarify them. Subtraction of the shared part of the test does not have any notable negative effect on the result and a value of such result rises significantly as seen on the later results.

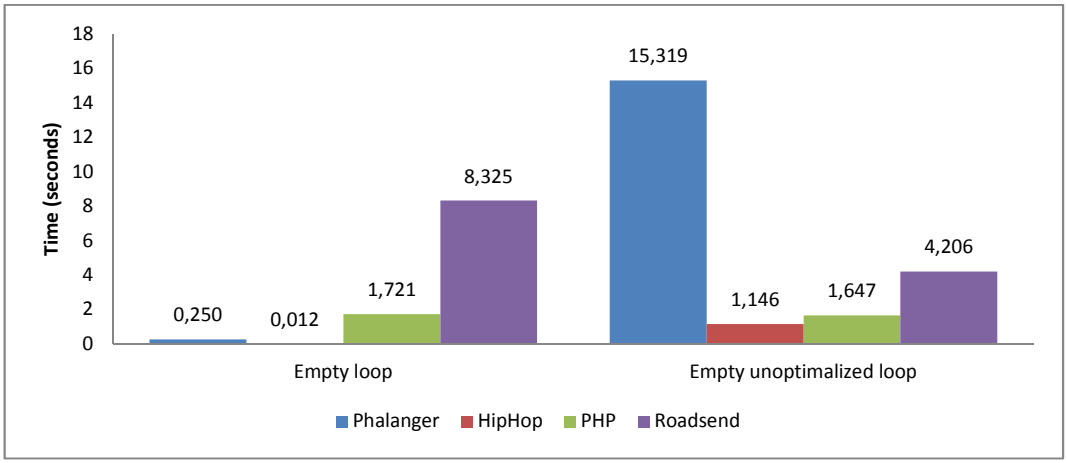


Figure 1 – Loop Benchmarks

The first benchmark represents the regular simple loop with directly accessed control variable, which is easy to optimize for compilers. In *PHP*, each variable has dynamic semantics and is stored in a hash table to enable indirect access by name. *Phalanger* is able to prove, that there is no operation capable of indirect access and therefore it is able to use a local variable instead of hash table. *HPHP* is additionally able to infer type of the control variable and remove its dynamic semantics. As noted before, we will subtract results of this benchmark from all following benchmarks of the same technology.

Unoptimized loop uses the control variable indirectly using a named variable. *HPHP* do not have the ability to infer variable type and remove dynamic semantics in this case and is two orders of magnitude slower than in the previous benchmark. *Phalanger* implements hash table representation very inefficiently since the slow dynamic binding is repeated each time the variable is accessed.

2.1.2 Static Fields

The following category of benchmarks measures access to static class fields. In single-threaded environment, static class field can be represented in the same way as a global static variable is, thus can be very fast. On the other hand in multithreaded environment, *TLS*[4] must be used for variable name. Access to *TLS* is somewhat slow because it is implemented by operating system and internally includes a map. Static class fields were introduced in PHP 5 and therefore are not supported by *Roadsend* compiler.

[4] *TLS* = *Thread-Local Storage* is a portion of memory, which stores variables for a single thread.

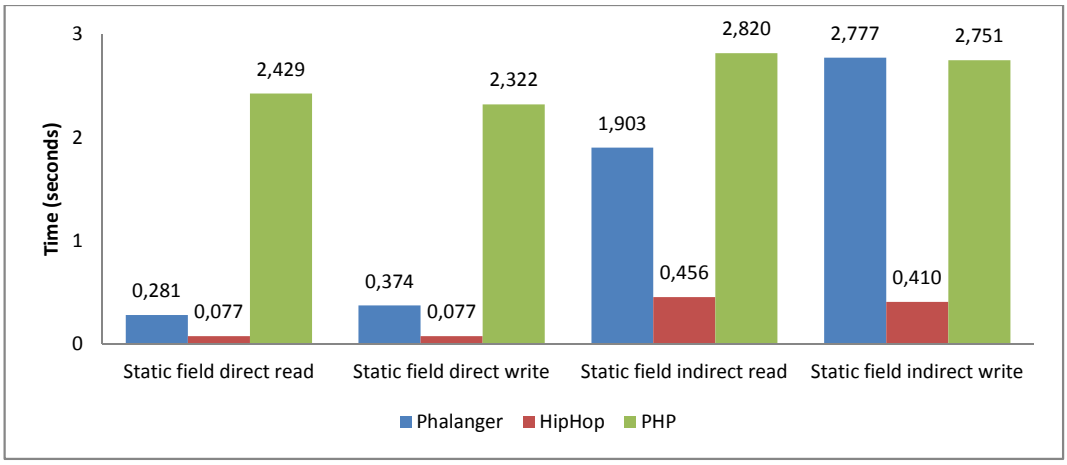


Figure 2 – Static Field Access Benchmarks

Read and write operations have roughly the same performance for both direct and indirect access for all technologies. In case of direct access, we can see that *HPHP* is faster than *Phalanger*, which we account to the fact that *HPHP* does not support *EVAL* operation and therefore set of static class fields is constant for a single compilation set and *HPHP* is able to exploit this by accessing the static variable directly without using *TLS*.

Phalanger implements the indirect access using *Phalanger's* dynamic runtime that first searches for containing type definition and then selects static field descriptor using it name. Then this descriptor dynamically emits a method that statically accesses the field. Truth to be told, this optimization does not perform very well. In HPHP, such access is done using a method that jumps using a jump table to access routine to the correct static field.

2.1.3 Declared Instance Fields

Declared instance field benchmarks use a reference to an object to access its field that has been declared in its class definition. In PHP, apart from marginal cases, declared fields only form a starting state of property hash table. However, compilers can easily exploit these declarations and optimize direct access to them, which is possible while still keeping a possibility for indirect access.

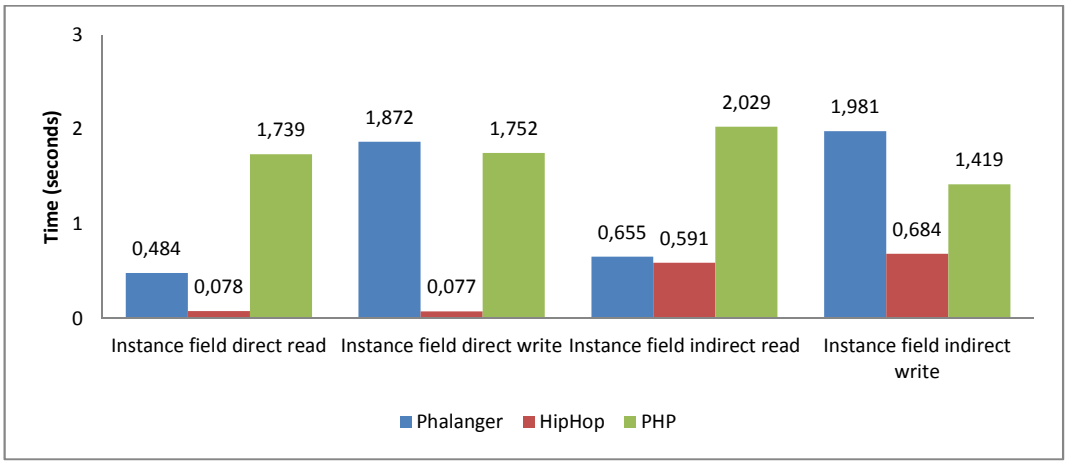


Figure 3 - Declared Instance Access Benchmarks

Phalanger uses a dynamic approach to access declared instance fields, which is very inefficiently implemented, and thus the advantage of compiled code is lost. This dynamic approach is needed since the compiler is not able to infer type of variable that stores reference to the object. [5]

HPHP on the other hand is able to infer the type is able to use highly optimized indirect access and static direct access that has a similar performance to static field direct access. *Roadsend* uses similar approach as HPHP, but inefficiently - we deduce this fact from even worse results of following benchmarks.

2.1.4 Undeclared Instance Fields

Undeclared instance field benchmarks are different from the previous since the used properties are not declared in class definition. This is correct in *PHP* as non-existent properties are "attached" to internal hash table of such properties. This complicates job for a compiler since even if we know a type we are accessing, it would need to add this field to the class definition, possibly leading to combinatorial explosion.

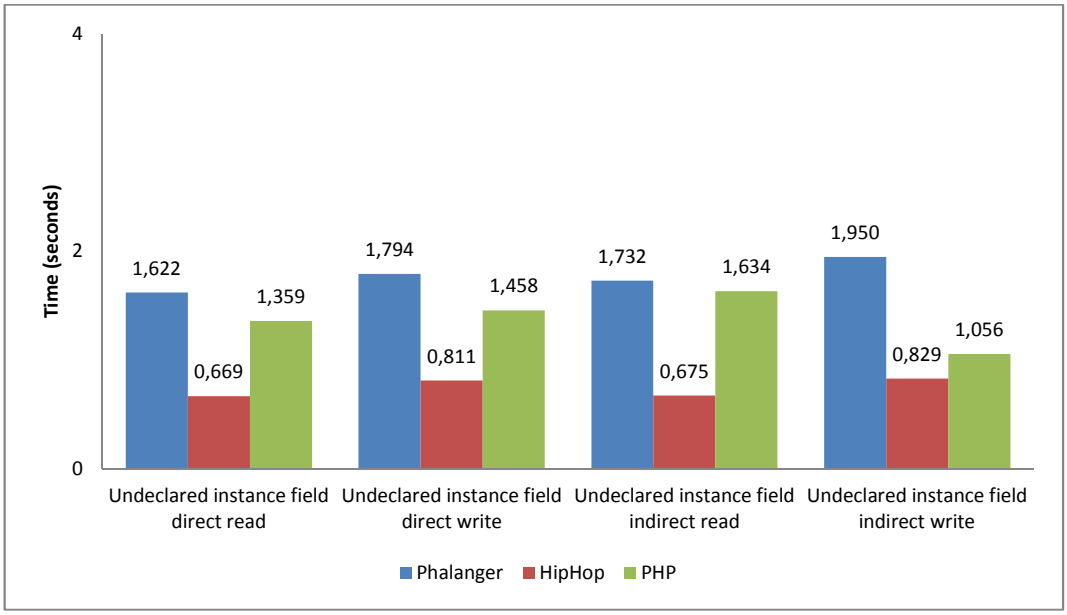


Figure 4 - Undeclared Instance Access Benchmarks

This principle is seen on Figure 4 as *HPHP's* performance drops to the level on indirect access to declared field both in direct and indirect case. As seen, all compilers use the completely dynamic access to the field.

2.1.5 Inherited Instance Fields

Another problem is inheritance relation between classes. For *PHP*, where fields are stored within a hash table shared between all classes on the inheritance path, there is no difference in performance for different declaring classes. In case of compilers, a field accessed from a class one or more levels lower in the inheritance hierarchy than it is declared must be searched for in the hierarchy path.

[5] This was improved by using *DLR* to cache fields resolutions and current Phalanger is about three times faster on all of these tests.

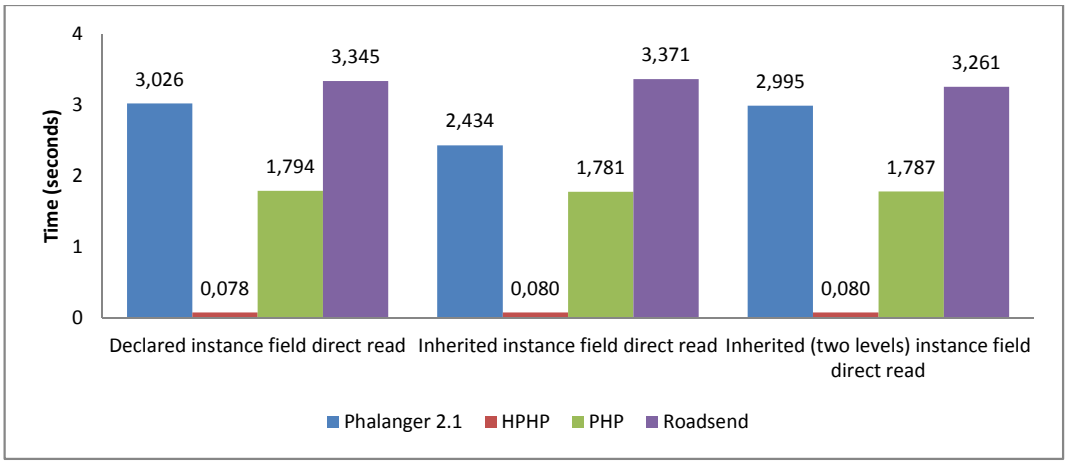


Figure 5 - Field Inheritance Benchmarks

Obviously, all technologies are independent on inheritance level. In case of *HPHP*, all accesses are completely static. *Phalanger* adds additional overhead because of the runtime resolution of field.[6]

2.1.6 Static Methods

Invocation of static method is another example of an operation that is easy to optimize by a compiler. If the target class is known to a compiler in point of static method call, it can emit simple call to this function in the same way that compilers do for static languages.

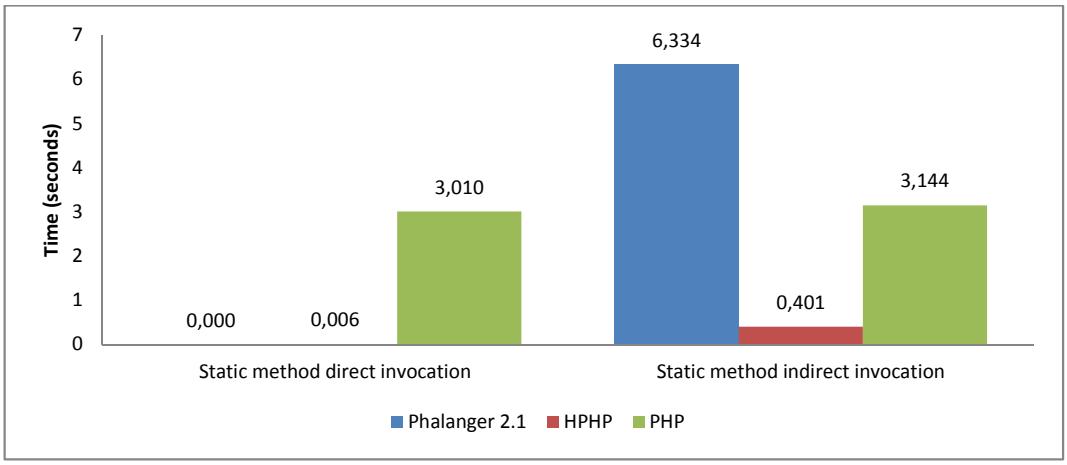


Figure 6 - Static Method Benchmarks

We can see this in Figure 6 in case of *Phalanger* and *HPHP* for which the performance cost associated with the direct static method call is negligible or zero because of optimization done by *JIT* and *C++* compiler respectively. Yet again, Phalanger is very slow in indirect invocation.

2.1.7 Instance Methods

Invocation of instance method benchmarks consist of a single object, method of which is called in direct and indirect way. In order to optimize this operation, a compiler needs to infer type of variable which contains the object and in case of indirect invocation, identify whether the name of the function is constant for the call site.

[6] Again, all tests were improved about five times by incorporating *DLR* mechanisms.

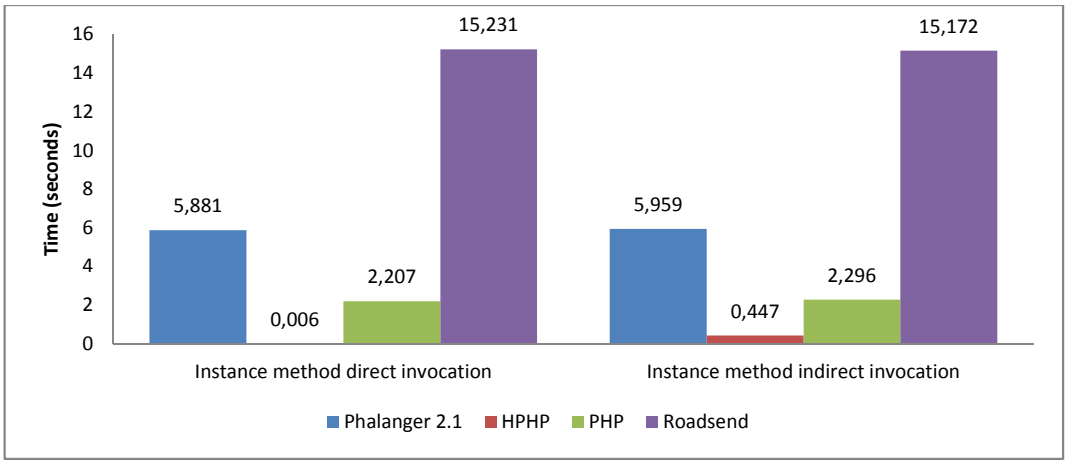


Figure 7 - Instance Method Call Benchmarks

Figure 7 shows us, that *HPHP* is able to infer type and call the method completely statically. In case of indirect it is apparent that the compiler is not able to propagate constant name of the function into indirect method call and uses dynamic dispatch to call the function. *Phalanger* uses dynamic dispatch, implementation of which is even slower than interpreted *PHP*. [7]

2.1.8 Inherited Instance Methods

Similarly to inherited instance fields we will also benchmark inherited instance methods. In these tests, there is hierarchy of three classes and we are testing performance of direct method invocation on each level. If a compiler is unable to resolve type of the called object, it needs to fall back into dynamic dispatch that can be slower when inheritance level rises.

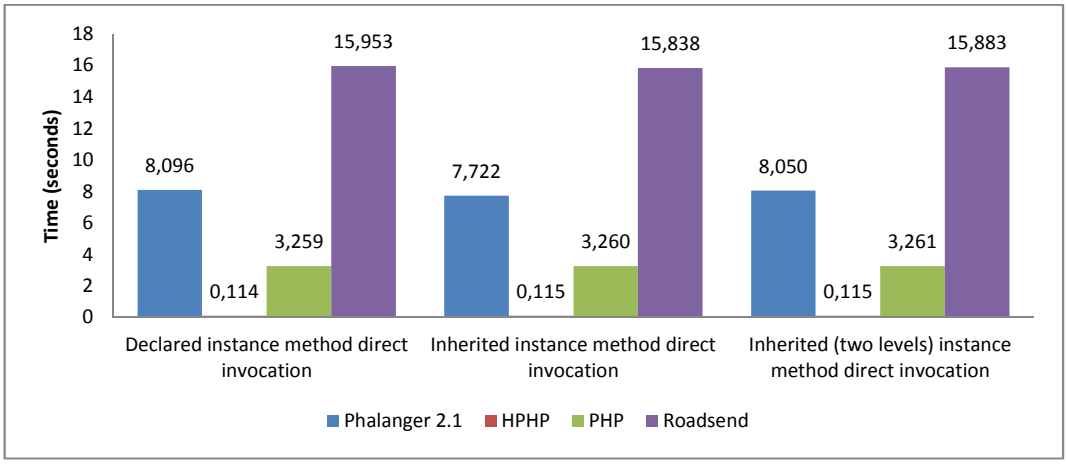


Figure 8 - Inherited Instance Method Benchmarks

Amount of work needed by *Phalanger*-compiled program rises in case of higher inheritance level (refer to previous set of benchmarks where the inheritance level is one), but regardless of the class that actually contains the method. Since all methods in *PHP* are virtual *HPHP* executes these benchmarks bit slower than without inheritance hierarchy.

[7] Yet again, this operation was improved by DLR that was incorporated into Phalanger after these benchmarks were done.

2.1.9 Operators

Operator benchmarks are targeted to measure performance of mostly used operators – addition and string concatenation. While other arithmetic operator would have similar performance as addition since their type semantics are very similar, string concatenation is non-constant operation that is used very often in a web application as its output is text. Addition test is implemented by incrementing a variable by one in a cycle.

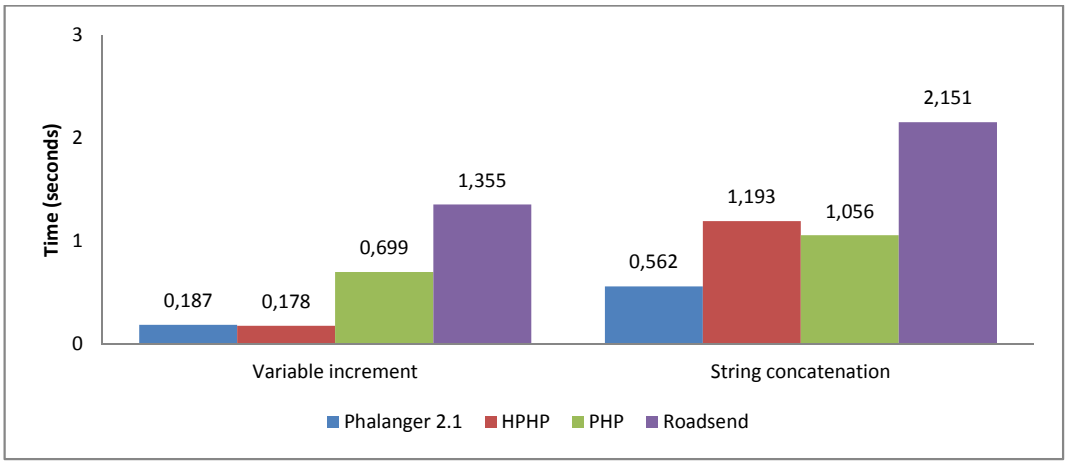


Figure 9 - Operator Benchmarks

Both *Phalanger* and *HPHP* utilize dynamic semantics that are faster than *PHP* because the code is not interpreted. As a side note, *HPHP* is able to infer type of a control variable of the for loop that contains the test as we have noted before, but it is not in case of variable incremented in the body of the for loop. The difficulty doing so lies in *PHP* number semantics in which integer that overflows becomes a floating point value. In case of for cycle control variable, *HPHP* uses a pattern that enables it to type the cycle control variable when it is bounded by two constant values and not assigned in the for cycle body. Doing this in arbitrary case is not simple and is usually done only for specific patterns as *HPHP* does.

In case of string concatenation, *PHP* and *HPHP* have nearly the same performance while *Phalanger* is two times faster than both. Since major part of string concatenation is memory reallocation, we attribute this difference to the way how *.NET* allocates memory – in garbage collected environment; memory can be allocated by simply incrementing one pointer without the need to search for a free chunk of memory.

2.1.10 Arrays

Objects are present in *PHP* since version 4.0 and before that, *PHP* arrays were used as a basic data structure as their semantics and library support allow to use them as arrays, hash tables, stacks, queues, records etc. Many programmers that were used to *PHP* before version 4.0 (which was released in 2000) or develop codebases intended for *PHP 3* still use arrays in place of objects. Additionally since *PHP* arrays are not simply arrays per se, they support both integer and string keys and arbitrary values. This complicates optimization in arbitrary case.

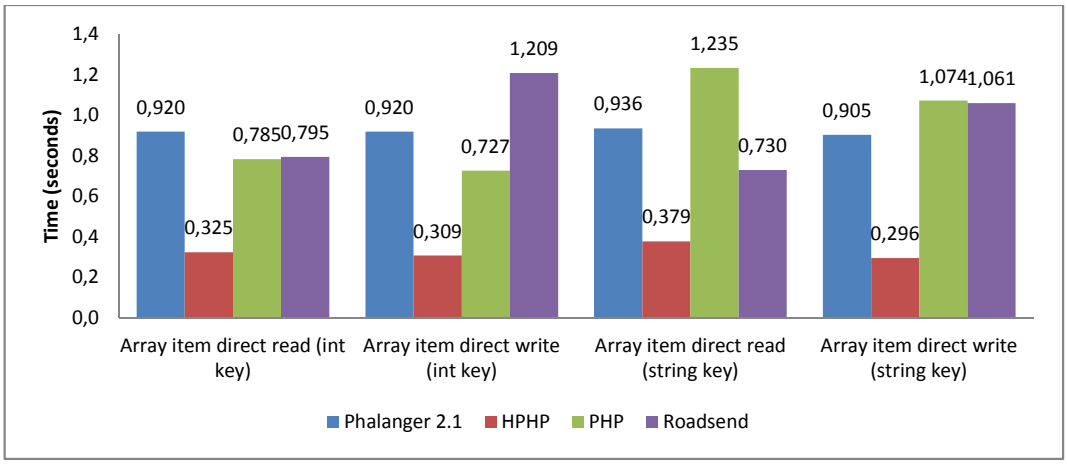


Figure 10 - Array Benchmarks

HPHP is faster than *Phalanger* but the difference is in line with other hash table performance differences we have observed before. In case of *PHP*, we can see that string key is somewhat slower than integer key. *Roadsend* benchmarks show performance difference between read and write, pointing to different hash table implementation.

2.1.11 Instantiation

Another important operation is object instantiation (*new* operator). To instantiate the object statically a compiler needs to be able to early bind class name. In static languages this is automatic, but in *PHP* it is not guaranteed that the class will declared before the instantiation is encountered. This requires a kind of control flow analysis to solve the arbitrary problem.

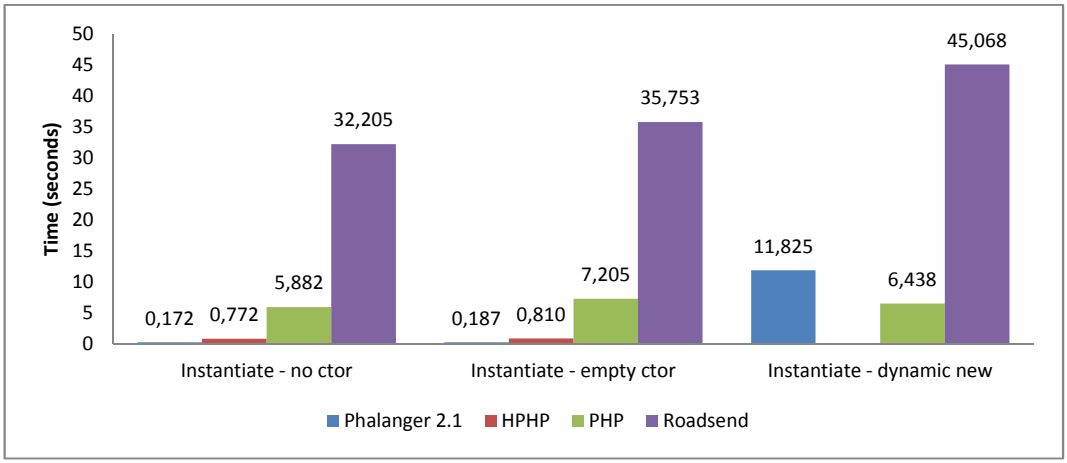


Figure 11 - Instantiation Benchmarks

Phalanger uses script inclusion graph that maps inclusions between script files and record a set of unconditional declarations for each script file (we have described this in more detail in section 1.4.3). In this way, the compiler is able to determine in most of typical cases which class the new operator should instantiate and emit static instantiation of it. In case we force the compiler to use dynamic instantiation, *Phalanger* is very slow as in all dynamic operations.

HPHP is able to determine the actual class using control flow analysis or similar analysis. However the instantiation is slower than in *Phalanger* which we again account to different type of memory

initialization and probably due to different representation of objects or aggressive *JIT* optimization. Performance of *PHP* is equal in all cases, since *PHP* does late binding of the class type each time.

2.1.12 A Note on Global Functions

We haven't tested global functions in separate test. However, for analysis, global functions are similar to class static fields, as classes are global declarations whose validity is dependent on global control flow, which is in *Phalanger* approximated by script inclusion graph analysis. Therefore, as we do not supply these test, their results would be similar to static method benchmarks, since both *Phalanger* and *HPHP* are able to early bind the call in most cases.

2.2 Application Analysis

Next stage of the performance analysis is analysis of real-world applications, which would allow us to tell which operations are important and which are marginally used. We have performed analysis of two real-world applications – *WordPress* and *MediaWiki*. These are probably the most used open-source *PHP* applications. We have planned to include analysis of two proprietary enterprise application we have been working with while optimizing *Phalanger*, but haven't been able to obtain permission to do so from company that holds the copyright.

We have briefly analyzed application source code using a performance profiler, a program that collects statistics of program execution including measurement of time spend within each method. Thanks to this information, we are able to approximately derive the distribution of each language feature as tested in micro benchmarks in total execution time of benchmarked application. This information helped us understand the real performance bottlenecks. We have used *JetBrains dotTrace 4.5.1 Performance* to obtain the results but we chose not to include

The test itself was performed using *Microsoft Visual Studio 2010 Load Test* that allowed us to test the highly concurrent environment that simulates real world scenarios in which Phalanger is typically used. We won't be providing detailed results and methodology, since the analysis was very extensive and requires in-depth knowledge of *Phalanger* and target applications. Moreover, these results are merely for orientation as performance profiler introduces a considerable and hard to identify statistical bias, since it changes the code of the application by adding profiling instructions. This disables some local optimizations performed both by processor and JIT compiler.

2.2.1 Library Functions and Emitted Code

In the first analysis, we have focused on determining how much of the time spent in an application is actually spent in the code emitted by *Phalanger*. The main motivation is whether the compiler poses a major burden to performance or whether the class library is not optimal. Note that it is virtually impossible to tell exact results and therefore we will be mentioning minimal ratio of PHP code performance that we were able to experimentally prove.

We are also including runtime functions that are related to dynamic behavior. We are specifically not including functions that are related to runtime types but are not possible to optimize by the compiler.

In case of *WordPress,* we have concluded that minimum of 29% of time is spend in code emitted by Phalanger when there is no data in the database. When we have filled the data with 50 testing pages separated into 5 categories, this ratio dropped to 17%. This change is attributed to regular expression evaluation that is used for *WordPress* output rendering and which is inefficient in .NET.

In case of MediaWiki, we have concluded that a minimum of 34% of time is spent in code generated by Phalanger. This ratio dropped to 28% when we have filled the testing site with 50 testing pages. Again this is mainly due to regex processing of the data, which is not used as much as in WordPress.

As a side note, the enterprise application we have mentioned earlier spends 50%-60% of the time within emitted code, since it has lot more complicated business logic and uses objects that are implemented inefficiently in Phalanger. In fact, when filled with more data, this ration has raised to 60%-70% Note that these results were assessed on older version of Phalanger and may not be up-to-date. Despite this, we are providing it for reference.

2.2.2 Distribution of Operations

The next step was identifying which operations are important within the results of emitted code mentioned earlier. We have divided these results to following categories:

- **Operators.** Since *Phalanger* does not emit any operators directly and rather uses functions implemented in *C#*, identifying time spend in operators was done using a sum of all operator functions.
- **Conversions.** Runtime conversions are necessary because PHP incorporated so-called type juggling, i.e. values being implicitly converted to the needed type.
- **Arrays.** *Phalanger* uses operators for handling arrays. However, some intensive functions are handled directly by *PhpArray* class.
- **Dynamic Variable Access.** Dynamic variable access performs a hash table lookup, which can be very time-consuming. In most cases, Phalanger is able to optimize local variables, but not global variables.
- **Emitted Code.** The rest of the time we attribute to emitted CIL byte code and operations needed for preparing dynamic dispatch etc.

We have grouped the results to include both applications because the distribution was very similar. Note that this distribution is for orientation only and will help us to design optimizations, not to obtain a precise distribution of operations. Again, obtaining precise results would be very complicated and would form a diplo thesis of its own, given the complexity of analyzed applications and code.

We have concluded following results:

- Operators – ~31% (~8.5% of total time)
- Conversions – ~10% (~2.8% of total time)
- Arrays – ~19% (~5.3% of total time)
- Dynamic Variable Access – ~18% (~5.0% of total time)
- Emitted Code – ~22% (~6.2% of total time)

The proprietary application that we have mentioned before spent around 60% of its non-library time by calling object methods and accessing object properties. Similarly to the results mentioned earlier, this result is not precise and up-to-date, but it illustrates the situation in object-oriented programs. Since Phalanger does not optimize calls to instance functions and methods, these results are believable even in current version.

2.3 Summary

Results presented in this chapter can be characterized by the fact that the least optimized and dynamic features of Phalanger pose the largest performance penalty. Additionally, if analyzed without context, this performance penalty is softened by the fact that most of the time is spend in library functions and only around 30% is taken by actual emitted code. This correlates with the fact that when *HPHP* compiles the static PHP code into executable, around 20% of the processor time is saved.

Our results indicate that when we remove several dynamic operations, we will be able save the same amount as *HPHP* does, or even more. Especially, this applies to object access, array access and dynamic dispatch. There are two ways of optimizations – first directly improve the dynamic operation, or second, remove the dynamic operation completely. In case of *PHP*, most dynamic operations can be removed by having type information about variables.

Another area of possible improvement is optimization of advanced type analysis, i.e. inference of type constraints that can be applied to an instance of type. This is especially needed for arrays, that have semantics of polymorphic hash table in PHP although they are most often used as regular array, i.e. with integer indices which range from *0* to *N* and single type of values. We will aim the rest of this thesis into inferring variable types and advanced type analysis.

3 Control Flow Analysis

Static analysis and resulting optimizations discussed in this work are centered on type analysis in PHP language as a main tool for the compiler to make inherently dynamic operations static. Type analysis infers valid typing information of untyped variables, which is used to remove dynamic operations from resulting object code. There are many implications of knowing typing of variables – removal or limitation of dynamic dispatch, improving performance of implicit type conversions or function type specialization. To implement type analysis efficiently, we need to have information about ordering of assign statements in the resulting program. Since a program can contain several types of loops, etc. we need to have a representation that would relieve us of such language-specific constructs.

Analysis of program's flow of control – possible ordering of program's statement execution – targets to group the statements that are always executed in a specific order, called *basic blocks*, and then build an oriented transition graph between these blocks, called *control flow graph* (*CFG*). This graph represents all branching and joins within the program and serves as a basis for most other analyses that are flow-sensitive. For more details about the general control flow analysis, the reader should refer to [**31**] as we are going to present algorithm adapted to the needs of Phalanger and PHP language.

3.1 General Requirements

For brevity, we will not represent all features of *PHP* language, especially exception handling statement and *goto* statements. Exception handling would require a specific set of nodes and additional logic that would complicate later algorithms. *Goto* statements would only change structure of the *CFG* and would complicate description of algorithms that we will include. Additionally these constructs are not often used in typical PHP programs.

3.1.1 Scope of Control Flow Analysis

We can differentiate between *intra-procedural CFA* and *inter-procedural CFA*. In intra-procedural *CFA*, we are interested only in local function context, taking calls to other functions as regular operations within the basic block. However, for dynamic languages as PHP, each call is a more complicated operation.

In a static language, we know specifically the callee, save for virtual methods. Additionally, the return values are strongly typed in a way the program needs it (this applies also for virtual methods). In case of virtual methods, we can identify the virtual table slot and the virtual call itself does not do a bigger overhead. Even then, static language compilers use various approaches to inter-procedural control flow and data-flow analyses to collect information needed for more aggressive optimizations. Compilers of dynamic languages are in a different position since declarations are not flow-insensitive and function return values are not typed. So, without further analysis, each call site is inherently dynamic and it is up to compiler to determine the target of the call and handle the untyped value the function returns.

In brevity, the inter-procedural control flow analysis (e.g. [**32**] [**33**]) targets to build a call graph of functions, i.e. a graph with functions as vertices and the relation of presence of call from caller to callee forming set of edges. In a static language, it is a matter of searching for all call sites and using results of semantic analysis to find the caller and thee callee. However, in a dynamic language the

actual callee is flow-sensitive, i.e. it depends on all paths of execution before the call site is approached, since the declaration of the callee could be conditional or could have not happened at all. In an arbitrary case this means that the analysis needs to take in account each path on which the caller can be called.

3.1.2 PHP Entry Points

A typical PHP program is intended to run as a web application. Such applications are built in a way that different requests are handled by different script files, thus having multiple entry scripts. This poses a problem since as a script can include several other scripts, it may result in completely different final program for each entry script file since the order of inclusions may vary.

PHP does not specify which scripts may be explicitly requested and each one script file has to be treated as a possible entry point if it is accessible from the web. Hence, in an arbitrary case, we need to compile N programs with N scripts in each, thus compiling $O(N^2)$ scripts in total in order to get maximum optimization performance. This is not an acceptable solution, since many applications contain several hundreds of scripts, thus slowing the compilation by several hundred times and growing the resulting binary by the same factor.

Solution to this is limiting the number of entry points by specifying an explicit list of scripts that can be entry points. Logically, many scripts are not intended to be entry points and an inhibition of any of these becoming an entry point is enforced on web-server level or using a generic error returned by such scripts. After this specification is made, we can proceed with two approaches:

- First, compile the program for each entry point independently.
- Or second, define a single virtual entry point which would include the script file based on the request.

Since the conditional inclusion can happen in regular scripts, in our opinion, the second approach is more viable because the arbitrary algorithm handling conditional inclusions would solve both problems – i.e. conditional inclusions and multiple entry points. After the program is analyzed in this way, we can remove the virtual script file since it would be replaced by part of the runtime or the web server.

Compiling the program independently for each entry point is problematic since we again perform the compilation several times increasing the time needed and size of the resulting binary file. A possible solution would be to find the common parts of the program, but we deem such algorithm very complicated and even then there will be need to deal with conditional inclusions separately.

3.1.3 Flow-sensitive Declarations

The fact that declarations of entities (functions and classes) are flow-sensitive poses a problem - a declaration can be conditional, i.e. be declared in a conditional branch of a program. Therefore we will henceforward differentiate between declaration and definition. A declaration is a statement that changes the set of valid declarations or in other words, since the declaration in PHP cannot be undone, it adds a new declaration to this set or throws a fatal error if a declaration with the same name was already present. A definition is a single version of the entity, usually one for each declaration statement.

We will try to decide for each AST node that references any entity, what definitions are valid for this node, i.e. what definitions occur on paths through the flow graph that start at en entry point and go through the node to any exit point. Usually there will be a small set of valid definitions and in most cases even only one. This is similar to reaching definitions algorithm that is oriented on variable values but can be used for element definitions as well.

When we will be analyzing variable types, we will need to identify returning value based on the context of the function caller. The approach mentioned above will be useful, since we will incorporate flow graphs of functions and methods to global flow graph, allowing the subsequent analyses to perform their operation even on these subprograms.

Other problem is presence of dynamic operations, such as *EVAL*, which can make any declaration. However, this apparent problem does not pose any difficulty for deciding what entity definitions are valid, because of the behavior of PHP's declaration mentioned before. There are two such situations:

- A declaration of entity E dominates the eval statement S_E, i.e. some definition of E is always valid on all paths through S_E before S_E.
- A declaration of entity E postdominates the eval statement S_E, i.e. some definition of E is always valid on all paths through S_E after S_E.

In the first case, if S_E makes another declaration of already declared E, an error occurs and validity of declaration that was already made is not violated. In the second case, we need to make sure that if S_E declared E the program would behave correctly – thus all references to E that would otherwise throw an error, have to behave as they should and call the definition of E declared by S_E. Then when a previously known definition of E is approached, and error should be throws. Thus in first case, the eval operation does not affect the validity of E at all and in the second case it does affect only the paths between S_E and declaration statements of E.

Other situations, when a declaration is not defined for all paths of the program are generally erroneous and we will not treat them in any way. Therefore, dynamic operations do not pose problem to deciding validity of declarations.

3.2 Control Flow Graph for Syntax Trees

Standard control flow graph uses basic blocks as nodes, i.e. maximum sequence of instructions without any branching with a branch instruction being the last. This is valid for linear representations where higher constructs of the language were transformed into a more low-level representation, such as in *Three Address Code* (*TAC*). In Phalanger, there is no such representation as the compiler does not need to deal with low-level optimization and we are presenting analysis done directly on the AST. The problem in AST is that within a single expression, there might be a conditional expression or a function call.

Because we are analyzing the control flow globally, we need to treat function call as a hierarchical node in the flow graph. Similarly, since type analysis is performed using the flow graph, the conditional expression needs to be treated as a branching node. The control-flow graph will occasionally break single expression into several basic blocks, based on the actual sequence of evaluation.

The inherent problem of dual representations is implementing most of the logic multiple times and growing memory requirements. We have tried to address this by reusing as much of the AST as we could, building the CFG that reuses most of non-branching a non-call statements and expression.

Along with Control Flow Graph definition we will include a description of flow charts, which will represent these control flow graph graphically. This will later help us to describe algorithms that work on the flow graph.

3.2.1 Virtual Variables

In order express sub-expression evaluation that is done in a well-defined order, we will need to introduce notion of virtual variables. These variables are only meant for control flow graph and are not named. We don't generally need to introduce local temporary variables since CIL is stack-based and it provides such variables automatically. Additionally the goal of CFA is not to change AST but rather provide alternate representation of it.

```
1:      $a = 1;
2:      $b = 2;
3:      $c = foo($a + $b, $b);  // #1 <- $a + $b, #2 <- $b, #3 = foo(#1,#2)
```

Snippet 7 – Use of Virtual Variables

Snippet 7 demonstrates the situation when virtual variables are needed. In the PHP language semantics, expressions "$a+$b" and "$b" are evaluated before the function call in that order. However, the call to "foo" function is dependent on these variables. Similarly the assignment to "$c" variable is dependent on return value of "foo".

Our representation of the control flow graph therefore includes so-called virtual statements, which allow us to express order of evaluation of expressions that are used in statements that form nodes of Control Flow Graph. These statements assign expression values to numbered virtual variables, which are passed along the edges of the control flow graph (and always have limited lifespan). In reality those virtual variables are implemented using a hash table that maps expressions to results of control flow node analysis and control flow nodes that reference those expressions in reality only query this hash table for already-collected results for sub-expressions.

3.2.2 Categories of Control Flow Nodes

We generalize the flow graph by introducing above described virtual statements that form part of linear nodes (which is our denomination for basic blocks). Thus, it can be said that edges of the flow graph pass values from originating node to target node. This principle will allow data-flow analysis and other analyses to fluently work with the control flow graph without need to preprocess it or without need to transform the whole AST into a flow graph.

We define a flow graph for each subroutine, which is in case of PHP a script, function or class method. Each control flow graph consists of four different categories of nodes:

- Special nodes that represent entry point, exit point and return value of a program, script or function.
- Linear nodes, which have arbitrary number of inbound transitions and one outbound transition. These nodes contain sequence of assign statements that assign expressions to

real variables, followed by a sequence of virtual assign statement that assigns expressions to virtual variables.

- Branching nodes, which have one inbound transition and set of outbound transitions representing the individual branches. Typically, a branching node is assigned a virtual variable upon which the decision of which branch will become active is made.
- Hierarchical nodes that have one inbound transition and one outbound transition. These hierarchical nodes specify their argument using virtual variables, including the identifier of target hierarchy (i.e. function name or script path), and their return value which will be stored in specific virtual variable.
- Empty nodes that are used solely for transformation purposes and are afterwards removed.

This design allow us to reuse maximum portion of the AST, especially the large and complicated expressions and assign statements, while expressing the change of control using different types of nodes.

3.2.3 Special Nodes

ENTRY node represents an entry point to a program, script, or a function. Each ENTRY node specifies a number of input arguments with their local variable names. It has exactly one outbound transition. We can see representation of the ENTRY node in Figure 12, where two arguments are supplied to it and named "$a" and "$b".

Figure 12 - Entry Node

RETURN node is a special node that represents returning of a value. It specifies a virtual variable that represents and an expression AST node, which is returned by the return statement. We show the graphical notation of RETURN node in Figure 13.

Figure 13 - Return Node

EXIT node is optionally preceded by a RETURN node, if a value is returned. If that is omitted, it means that function, script or method does not return any value. EXIT node has a single inbound transition and no outbound transitions. Figure 14 shows representation of EXIT node in the flow chart.

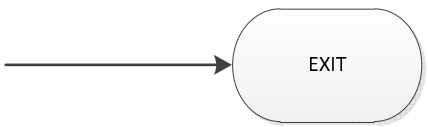


Figure 14 - Exit Node

3.2.4 Linear Nodes

LINEAR node represents all variable handling in the program. It has an arbitrary number of inbound transitions and exactly one outbound transition. In Figure 15 we show representation of a linear node in flow chart.

Figure 15 - Linear Node

LINEAR node is formed by two sequences – first is sequence of assign statements that assign an expression to a variable. Second is sequence of virtual statements, which assign expression value to a virtual variable. If any of expressions contains a function call or other non-linear operation, it is guaranteed by the transformation algorithm, that each of these expressions is evaluated prior to their use in the linear node. To express this, graphical notation replaces these expressions by virtual variables associated with them. In the real implementation, each item would be a real assign statement AST node and in case of virtual assign statement there will be reference to an expression AST node for each virtual assignment.

3.2.5 Branch Nodes

Single type of branch node - BR node, represents a test if a value is equal to TRUE Boolean value. A BR node has two outbound transitions – one TRUE case and one for FALSE case. Figure 16 show the representation of the branch node in the flow graph.

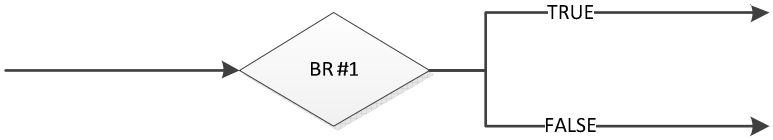


Figure 16 - Branch Node

Each BR node additionally specifies identifier of virtual variable that will be used for the test. This variable represents an expressions AST node that was already assigned in a virtual statement in preceding linear node.

3.2.6 Hierarchical Nodes

There are five types of hierarchical nodes – CALL, CALLOBJ, INCLUDE, NEWOBJ and EVAL nodes. All nodes have a single outbound transition and an arbitrary number of inbound transitions. All these nodes represent an underlying flow graph, is it known at compile time or not. An example of CALL node in a flow chart can be seen in Figure 17, where a call to global function with two arguments “#3” and “#4” is made. Variable “#2” represents a function name expression and variable “#” represents target virtual variable for return value.

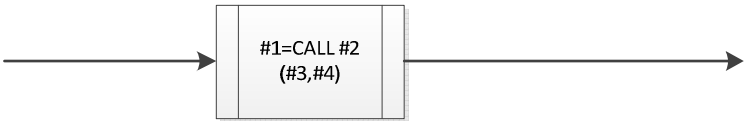


Figure 17 - Call Node

Similarly, a CALLOBJ node accepts arguments and function name and additionally a virtual variable representing a reference to the function. A NEWOBJ node specifies a virtual variable representing class name that it to be instantiated and arguments for the constructor. Both are seen in Figure 18.

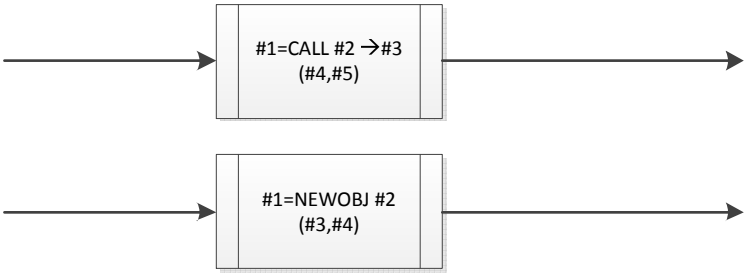


Figure 18 - CALLOBJ and NEWOBJ nodes.

INCLUDE node specifies a virtual variable which contains the name of the script that is to be included. Additionally it specifies behavior of the inclusion statement – "include", "include once", "require" and "require once". An example of INCLUDE node with "include once" behavior is shown in Figure 19.

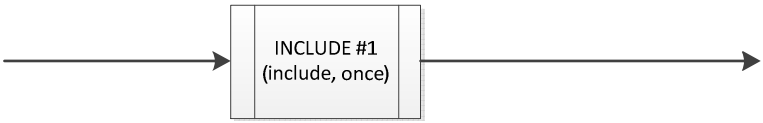


Figure 19 - Include Node

Last node mentioned, the EVAL, is a dynamic operation that can perform anything the language is able to express. However, in some situations, we will be able to prove that the argument is constant and therefore we will be able to replace the node by its flow graph.

3.3 Control Flow Graph Examples

In this section we will demonstrate several examples of how a source code is expressed using a Control Flow Graph we have previously defined. This will help us explaining the algorithm for building the flow graph.

3.3.1 Virtual Variables

Since the hierarchical nodes can partition a single statement and especially expressions in the middle of their hierarchy, we will provide example how the resulting flow graph looks like. In order to express change of control to other procedures and prepare the flow graph for subsequent analyses, we need to pass virtual variables to nodes that are evaluated later. Consider following example:

```
1:      function qux($a,$b,$c,$d)
2:      {
3:              return foo(bar($a + $b, $c + $d), bar($a - $c, $b - $d));
4:      }
```

Snippet 8 - Multiple Functions In Expression

Snippet 8 demonstrates such situation. In order to evaluate result of function foo, we first need to call function bar twice and then pass the results to function foo. These results have only a limited scope, since they are not needed after call to foo is made. To express this without using a real variable that would complicate the following analysis, we name these result as #1, #2 and so on.

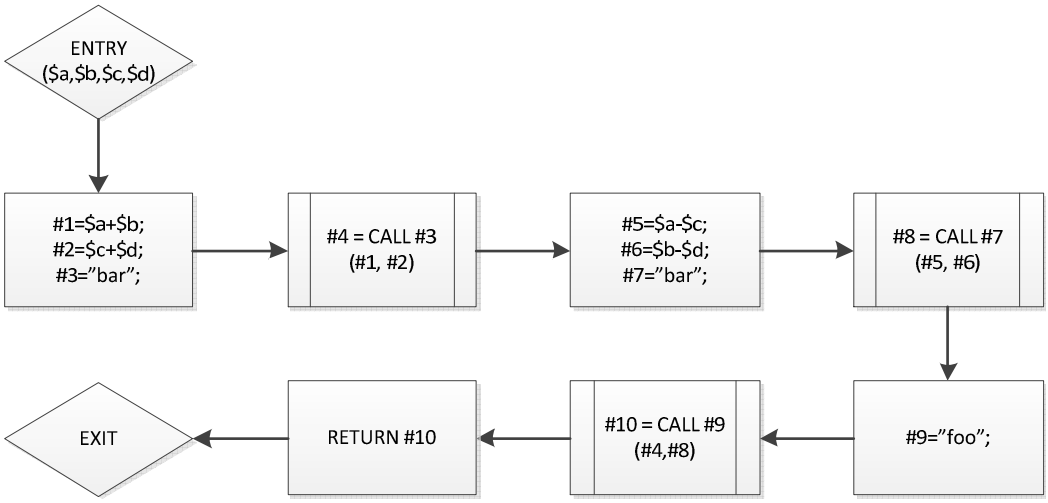


Figure 20 – Multiple Virtual Variables

Figure 20 demonstrates graphical representation of the function above. As we can see, the flow graph of the function from Snippet 7 contains 9 nodes. The original AST contains 18 nodes. Therefore, the memory requirements associated with the flow graph are actually lower than original AST, because we are reusing part of representation.

3.3.2 Conditional Expressions

Conditional expressions are not handled by common Control Flow Graphs since these representations are built on top linear representation that have such high-level constructs removed and replaced by more low-level representations.

```
1:      function min($a,$b)
2:      {
3:              return $a<$b ? $a : $b;
4:      }
```

Snippet 9 - Trivial Conditional Expression

In Snippet 9 we define a function that returns the lower of two supplied values. The control flow graph will use virtual variables in similar manner as temporary variables would be used, but will keep them distinct.

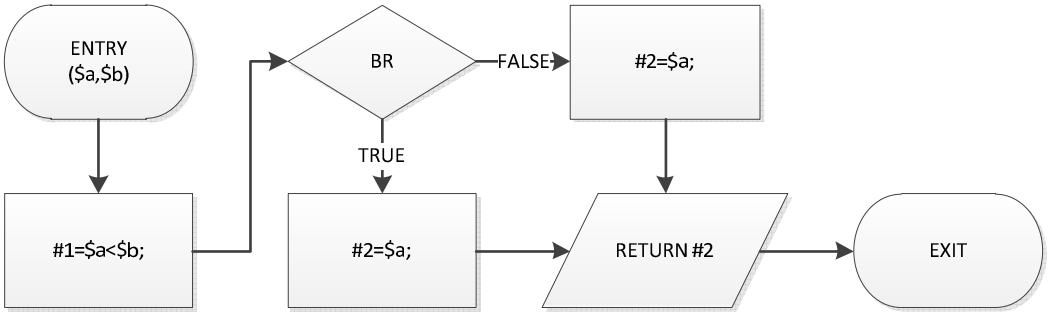


Figure 21- Conditional Expression Flow Graph

As seen in Figure 21, conditional statement uses a branch node, after which both possible values are assigned in two branches. The flow then joins in the return node that returns result of both branches.

3.3.3 Control Statements

We will demonstrate how a control flow graph with more control statements such as IF and WHILE loop looks like. In Example 1 we show code of such sample function that contains a cycle and if statement.

```
function foo($s, $n) {
        $c = 0;
        $x = “”;
        while (strlen($s) + $c < $n) {
                $x = $x . $s;
                $c = $c + strlen($s);
        }
        if ($c < $n)
                $x = $x . substr($s, 0, $n - $c);
        return $x;
}
```

Example 1 - Code of the function foo that will be used as a basis for sample flow graph.

This function is supplied with a pattern and required output length. It repeats the pattern so many times that the resulting string is long exactly as specified by the argument. The resulting flow graph contains all types of nodes described above as seen in Figure 22.

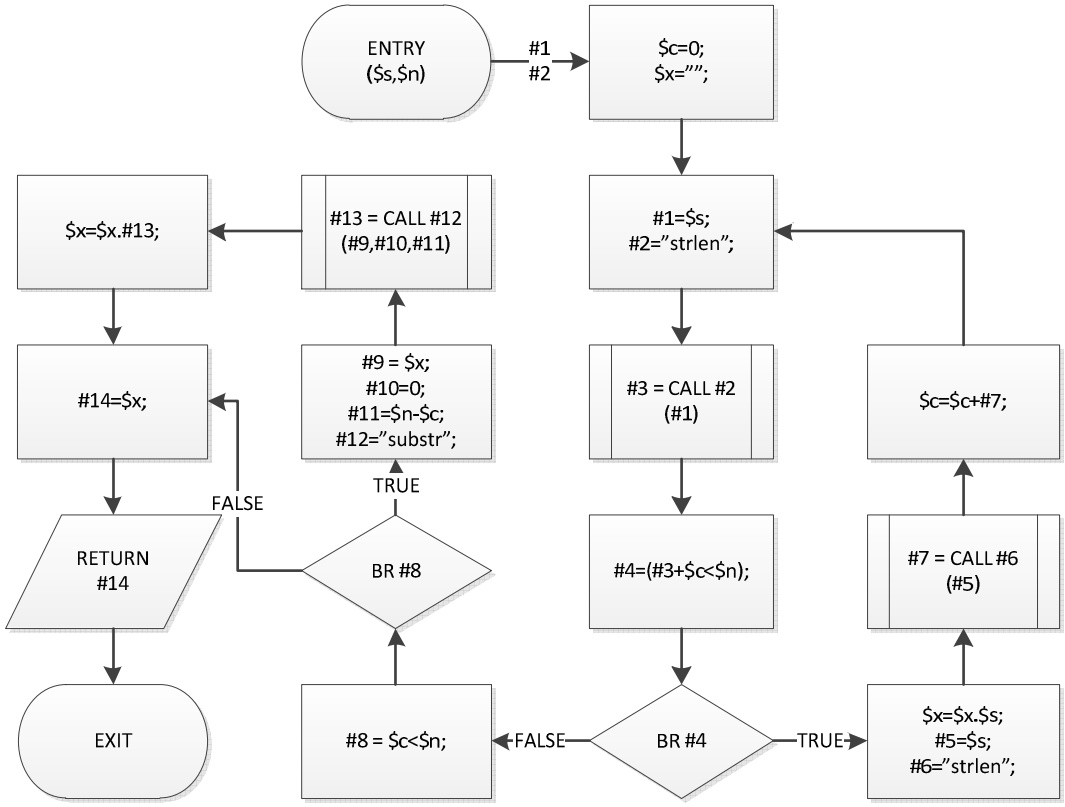


Figure 22 - Simple Function Flow Graph

If we will parse the source code given in Example 1, resulting AST would have approximately 45 AST nodes (depending on exact representation). The flow graph as seen in Figure 22 has 17 nodes, which is lower than the complexity of the AST. Since the flow graph partially reuses AST nodes, the additional memory cost associated with the flow graph would be around half of the cost of the AST.

3.4 Local Analysis

In the first stage of the control flow analysis we will prepare control flow graphs for each script, function or method separately. In addition we will create a flow graph for virtual entry point, which we have described earlier. The purpose of the local analysis is to create local control flow graphs which would be later used during both intra and inter procedural control flow and data flow analyses.

Algorithms are presented in pseudo code that is somewhat similar to OCaml family of languages [**34**] (but we use imperative syntax rather than functional) combined with notation of mathematical logic, set theory and pseudo predicates that are described in later text. We do not use Currying[8] as we want the code to be readable to reader unfamiliar with ML-style syntax. We are not providing definitions of functions and structures that are obvious to understand from the context. Additionally, we are not presenting how expressions are marked as virtual variables, because it is needed for data-flow algorithms and is realized in a much simplified way than it would be formalized.

[8] Currying (first described in [**47**]) is a function notation and definition style which expresses multi-argument functions as a "chain" of single argument functions, thus allowing partial specification of arguments. This notation style is used in Haskell, ML, F# and other functional languages.

We do not present transformation for whole set of AST nodes, but rather for minimum set, that consists of assign, block, if, while, break, continue and return statements and operator, variable, constant and call expressions. The algorithm does not produce maximal linear nodes (basic blocks) and does produce imaginary empty nodes. We introduce these changes for sake of simplicity and later informally describe how this flaw would be corrected.

3.4.1 Recursive Transformation Algorithm

Let us have a set N_{AST} of all possible AST nodes and set N_{CFG} of all possible CFG nodes and E_{CFG} a set of all possible edges of control flow graph. We define a structure $CFG_{PARTIAL}$ that represents a result of partial control flow analysis as specified in Pseudocode 1. This structure represents a part of the CFG, which is a result of transformation of a particular subtree. It recognizes a node in which the flow enters the subtree and a set of edges which mark a regular end of the subtree. Moreover, we specify sets of edges that are result of break statement, continue statement and return statement. These sets are handled by different transformation algorithms for specific nodes and in the end bound to some other node. For example, while statement binds continue edges to an entry node of the loop body.

$$
\begin{aligned}
&CFG_{PARTIAL} = \\
&\quad EntryNode: N_{CFG} \\
&\quad ExitEdges: \mathcal{P}(E_{CFG}) \\
&\quad BreakEdges: \mathcal{P}(E_{CFG}) \\
&\quad ContinueEdges: \mathcal{P}(E_{CFG}) \\
&\quad ReturnEdges: \mathcal{P}(E_{CFG})
\end{aligned}
$$

Pseudocode 1 – Definition of partial flow graph structure

We then define an algorithm $CFA(N_0 \in N_{AST}) \rightarrow N_{CFG} \times N_{CFG}$, where $N_0 \in N_{AST}$ is a root node of a function, script or method as described in Pseudocode 2. This algorithm defines an entry node and an exit node. Then, it evaluates the partial transformation algorithm on the script, function or method body, usually represented by a block statement. Resulting partial flow graph represents the flow within the block statement. We connect the $Next$ connection going out of the $Entry$ node to $R.EntryNode$, representing flow out of entry point of the function. Similarly we connect all $ExitEdges$ to $Exit$ node, representing implicit return, and $ReturnEdges$ to $Exit$ node, representing explicit returns.

$$
\begin{aligned}
&Let\ CFA(N_0 \in Script \cup Function \cup Method) \rightarrow N_{CFG} \times N_{CFG} = \\
&\quad Entry \leftarrow new\ EntryNode \\
&\quad Exit \leftarrow new\ ExitNode \\
&\quad R \leftarrow CFA_{PARTIAL}(N_0) \\
&\quad Connect(Entry.Next, R.EntryNode) \\
&\quad For\ each\ e \in R.ExitEdges\ do \\
&\qquad Connect(e, Exit) \\
&\quad For\ each\ e \in R.ReturnEdges\ do \\
&\qquad Connect(e, Exit) \\
&\quad Return\ (Entry, Exit)
\end{aligned}
$$

Pseudocode 2 – Definition of transformation algorithm

Algorithm $Connect(E_{CFG}, N_{CFG}) \rightarrow unit$ is a special algorithm that connects an unbound edge of the flow graph to a node. We won't we describing how such algorithm might work since it depends on the particular implementation as is out of scope of this chapter.

We define algorithm $CFA_{PARTIAL}(N \in Statement \subseteq N_{AST}) \rightarrow CFG_{PARTIAL}$ for an AST node $N \in Statement$ that represents a statements in a polymorphic way, i.e. with different definition for each subtype of $Statement$. This algorithm transforms a given statement into its partial flow graph. Given nature of control flow graph formalization described earlier, all control statements are encoded using *Branch* and *Return* nodes with additional logic being hidden in the structure of the flow graph. Thus, only assign statements remain in linear nodes in the described model of AST.

For expressions, we define $CFA_{PARTIAL}(N \in Expression \subseteq N_{AST}, B \in Boolean) \rightarrow CFG_{PARTIAL}$ for AST node $N \in Expression$ that represents an expression. The Boolean argument specified whether the expression has to be automatically assigned to a virtual variable. This automatic assigning is present because of possible fragmentation of a single expression by several *Call* nodes. It is an auxiliary that automatically creates linear nodes out of function argument expressions.

3.4.2 Block Statement

Each block statement B consists of a sequence of statements, therefore we denote $B = (S_1, \dots, S_n)$ as a vector of statements. We define the control flow partial transformation of the block statement in the following way:

```
1:  Let CFA_PARTIAL(N ∈ Block = (S_1, ..., S_n)) → CFG_PARTIAL =
2:      R ← new CFG_PARTIAL
3:      k ← false
4:      R.EntryNode = new EmptyNode
5:      R.ExitNode = R.EntryNode.Next;
6:      For i = 1 to n do
7:          LS ← ∅
8:          l ← CFA_PARTIAL(S_i)
9:          R.BreakEdges = R.BreakEdge ∪ l.BreakEdges
10:         R.ContinueEdges = R.ContinueEdges ∪ l.ContinueEdges
11:         R.ReturnEdges = R.ReturnEdges ∪ l.ReturnEdges
12:         If k = false then
13:             R.EntryNode = l.EntryNode
14:         Else
15:             For each e ∈ R.ExitEdges do
16:                 Connect(e, l.EntryNode)
17:         R.ExitEdges = l.ExitEdges
18:     Return R
```

Pseudocode 3 - Transformation of Block AST node

The algorithm successively transforms inner statements of the block statement into partial flow graphs and serially connects them. All non-linear free edges are joined in the resulting partial flow graph. When a block statement contains only a single inner statement, the algorithm uses its exit edges as exit edges as exit edges of the resulting flow graph. Otherwise, exit edges are connected to

entry node of the next statement partial flow graph. We do not join consecutive linear nodes into a single node. This can be solved either by post processing the resulting flow graph or by alternating the algorithm to join the nodes.

3.4.3 If Statement

When transforming an IF statement into a control flow graph, we need first transform the expression and connect the result to partial flow graphs of true and false branch.

```
1:   Let CFA_PARTIAL(N ∈ If = (E,T,F)) → CFG_PARTIAL =
2:       R ← new CFG_PARTIAL
3:       e_r ← CFA_PARTIAL(E, true)
4:       t_r ← CFA_PARTIAL(T)
5:       f_r ← CFA_PARTIAL(F)
6:       b ← new BranchNode(E)
7:       For each e ∈ e_r.ExitEdges do
8:           Connect(e, b)
9:       Connect(b.True, t_r.EntryNode)
10:      Connect(b.False, f_r.EntryNode)
11:      R.EntryNode = e_r.EntryNode
12:      R.ExitEdges = t_r.ExitEdges ∪ f_r.ExitEdges
13:      R.BreakEdges = t_r.BreakEdges ∪ f_r.BreakEdges
14:      R.ContinueEdges = t_r.ContinueEdges ∪ f_r.ContinueEdges
15:      R.ReturnEdges = t_r.ReturnEdges ∪ f_r.ReturnEdges
16:      Return R
```

Pseudocode 4 – Transformation of If AST node

IF statement creates a node of the type $BranchNode$ that has two edges, which are accessed as its fields $True$ and $False$. Then, the condition expression and true and false statements are transformed and their partial flow graphs are connected to the branch node. Entry node is always the node representing an expression, while exit edges are union of exit edges of both true and false branch node. Break, continue and return edges from true and false branches are also joined in resulting partial flow graph.

3.4.4 While Statement

While statement we will provide transformation algorithm, which does not have any conditional and is therefore a natural cycle. This is equivalent as we support Break and Continue statements.

```
1:    Let CFA_PARTIAL(N ∈ While = (S)) → CFG_PARTIAL =
2:        R ← new CFG_PARTIAL
3:        s_r ← CFA_PARTIAL(S)
4:        R.EntryNode = s_r.EntryNode
5:        For each e ∈ s_r.ExitEdges do
6:            Connect(e, s_r.EntryNode)
7:        For each e ∈ s_r.ContinueEdges do
8:            Connect(e, s_r.EntryNode)
9:        R.ExitEdges = s_r.BreakEdges
10:       R.ReturnEdges = s_r.ReturnEdges
11:       Return R
```

Pseudocode 5 - Transformation of While AST Node

We can see that *While* transformation returns empty break and continue sets. It connects regular exit edges on the inner statement to the entry node, thus forming a loop. Break sets are passed as exit edges of the whole loop and continue edges are connected to the entry node.

3.4.5 Break Statement

Break statement is needed since while cycle we have described does not have a break condition. Such formalization is more similar to real-world PHP programs.

```
1:    Let CFA_PARTIAL(N ∈ Break) → CFG_PARTIAL =
2:        R ← new CFG_PARTIAL
3:        R.EntryNode = new EmptyNode
4:        R.BreakEdges = {R.EntryNode.Next}
5:        Return R
```

Pseudocode 6 - Transformation of Break AST Node

The transformation function creates auxiliary node that is later removed. This node acts as an entry node and the edge originating in it is returned as break edge.

3.4.6 Continue Statement

Continue statement is not needed for minimal set of language, but has the similar logic to break statement and is used often in imperative languages as PHP.

```
1:    Let CFA_PARTIAL(N ∈ Continue) → CFG_PARTIAL =
2:        R ← new CFG_PARTIAL
3:        R.EntryNode = new EmptyNode
4:        R.ContinueEdges = {R.EntryNode.Next}
5:        Return R
```

Pseudocode 7 - Transformation of Continue AST Node

Again, we use auxiliary empty node to simplify the algorithm and keep Exit, Break and Return edge sets empty.

3.4.7 Return Statement

The return statement is used for returning a value. It includes an expression that specifies the return value and has to be transformed beforehand to ensure that it does not contain any call node.

```
1:  Let CFA_PARTIAL(N ∈ Return = (E)) → CFG_PARTIAL =
2:      R ← new CFG_PARTIAL
3:      e_r ← CFA_PARTIAL(E, true)
4:      r ← new ReturnNode(E)
5:      For each e ∈ s_r.ExitEdges do Connect(e, r)
6:      R.EntryNode = e_r.EntryNode
7:      R.ReturnEdges = {r.Next}
8:      Return R
```

Pseudocode 8 - Transformation of Return AST Node

Return statement transforms the expression into a linear node with virtual assign statement which then passes control to return node.

3.4.8 Assign Statement

Assign statement creates a single linear node, in which the statement is put. The inner node is transformed by expression transform algorithm.

```
1:  Let CFA_PARTIAL(N ∈ Assign = (V, E)) → CFG_PARTIAL =
2:      R ← new CFG_PARTIAL
3:      e_r ← CFA_PARTIAL(E, false)
4:      l ← new LinearNode(V, E)
5:      If e_r.ExitEdges = ∅ then
6:          R.EntryNode = l
7:          R.ExitEdges = {l.Next}
8:      Else
9:          R.EntryNode = e_r
10:         For each e ∈ e_r.ExitEdges do Connect(e, l)
11:         R.ExitEdges = {l.Next}
12:     Return R
```

Pseudocode 9 - Transformation of Assign AST Node

In case the assigned expression contains any call expressions, non-empty graph will be returned by its transformation. If the graph is empty, the linear node created by this algorithm forms the whole resulting graph.

3.4.9 Expressions

Previously used definition of expression transformation accepts two arguments – AST node and Boolean value that informs the function whether it should wrap the transformed expression into a virtual assign statement. Since this functionality is shared by all expression transformation variants, we will provide a generic implementation.

```
1:  Let CFA_PARTIAL(E ∈ Expression, B ∈ Boolean) → CFG_PARTIAL =
2:      e_r ← CFA_PARTIAL(I)
3:      If B then
4:          R ← new CFG_PARTIAL
5:          l ← new LinearNode(new VirtualStatement(E))
6:          R.ExitEdges ← {l.Next}
7:          If e_r.ExitEdges ≠ ∅ then
8:              R.EntryNode ← e_r.EntryNode
9:              For each e ∈ e_r.ExitEdges do Connect(e, l)
10:         Else
11:             R.EntryNode ← l
12:         Return R
13:     Else
14:         Return e_r
```

Pseudocode 10 - Arbitrary Expression Transformation

3.4.10 Call Expression

Call expression is the main reason for different flow graph specification as our main goal is to support expansion of hierarchical nodes (which are similar to the CALL node which is created by the following transformation).

```
1:  Let CFA_PARTIAL(N ∈ Call = (I, A_1 … A_n)) → CFG_PARTIAL =
2:      R ← new CFG_PARTIAL
3:      i_r ← CFA_PARTIAL(I, true)
4:      R.EntryNode ← i_r
5:      E ← {i_r.Next}
6:      For each a_r ∈ {A_1, … , A_n} do
7:          For each e ∈ E do Connect(e, a_r)
8:          E ← a_r.ExitEdges
9:      l ← new CallNode(I, A_1 … A_n)
10:     For each e ∈ E do
11:         Connect(e, l)
12:     R.ExitNode ← l.Next
13:     Return R
```

Pseudocode 11 - Transformation of Call AST Node

Inner expressions are assigned to virtual variables and resulting partial flow graph are connected. Note that function identifier is also handled as an expression.

3.4.11 Operator Expressions

Operator expression types do not change the program's flow, but their sub-expressions are able to. These AST expressions represent operators that are defined by the language.

```
1:  Let CFA_PARTIAL(N ∈ Operator = (X, Y)) → CFG_PARTIAL =
2:      R ← new CFG_PARTIAL
3:      x_r ← CFA_PARTIAL(X, false)
4:      y_r ← CFA_PARTIAL(Y, false)
5:      If x_r.ExitEdges ≠ ∅ then
6:          R.EntryNode ← x_r.EntryNode
7:          If y_r.ExitEdges ≠ ∅ then
8:              For each e ∈ x_r.ExitEdges do
9:                  Connect(e, y_r.EntryNode)
10:             R.ExitEdges ← y_r.ExitEdges
11:         Else
12:             R.ExitEdges ← x_r.ExitEdges
13:     Else
14:         If y_r.ExitEdges ≠ ∅ then
15:             R.EntryNode ← y_r.EntryNode
16:             R.ExitEdges ← y_r.ExitEdges
17:         Else ()
18:     Return R
```

Pseudocode 12 - Transformation of Operator Expressions

Operator nodes are basic combinators of partial flow graphs of expressions. If both inner partial flow graphs are empty, an empty partial flow graph is also returned. If both partial flow graphs are not empty, right-hand operand flow graph is connected after left-hand operand flow graph. Otherwise, a non-empty partial flow graph is returned.

3.4.12 Variable and Constant Expressions

Variable expressions are one of the two types of leaves in our simplified model of control-flow analysis and are encoded by an empty partial flow graph. The same applies for constant expressions that represent constant values and are not influencing program's flow.

```
1:  Let CFA_PARTIAL(N ∈ Variable ∪ Constant) → CFG_PARTIAL =
2:      Return new CFG_PARTIAL
```

This empty value is then automatically treated by other expression-handling functions as demonstrated before.

3.5 Global Analysis

Global or intra-procedural control flow analysis is performed in our model by substituting CALL and other hierarchical nodes by flow graph of target definitions (methods, scripts or functions). This is not possible if the call is late bound. This means that results of control-flow analysis are dependent on results of other analyses and optimizations which in turn use control-flow graph to infer their results. This cyclical dependency of analyses can be solved by two approaches:

- **Iterative analysis**, which iterates the analyses until a fix-point is reached.
- **Combined analysis**, which combines all techniques into a single analysis.

As it is obvious, the combined analysis is very hard to implement and is stronger since it is able to solve cyclic dependency optimizations when two optimization steps are dependent one on another. Because of its complexity, we will not be considering this type of analysis composition further and we will concentrate on iterative analysis.

Considering again the whole PHP language, we are able to bind the call site (or inclusion site) to its definition when two conditions are met. First, the site must have a constant binding expression, which is usually met as non-constant binding expressions indicate indirect call or dynamic inclusion. Second, in all flows through the call site the declaration of the specified name must have the same definition. In case of script files, this is trivial since this happens only if the script file is located on disk during the compilation and dynamic inclusions are disabled (this is an option of Phalanger compiler that allows it to compile assemblies with limited dynamic behavior).

Global control flow analysis starts in a flow graph of virtual entry point that dispatches the control to all entry points specified for compilation. It perform depth first search on the current flow graph, entering the CALL and other hierarchical nodes when above specified conditions are met. This expansion is not performed recursively.

The actual guest analyses, especially the type analysis described later, can perform additional steps on the flow graph and request the expansion even for recursive nodes, but generally, global control flow analysis is not meant for anything more than building (or traversing) the global control flow graph, if possible.

3.6 Summary

We have described an efficient way of performing *control flow analysis* on *abstract syntax trees* of dynamic languages. In the following two chapters, we will use this algorithm intuitively, without using the formalization specified above for means of describing how the transformation algorithm works.

4 Type Analysis

This chapter discusses type analysis algorithm for PHP. It consists of three main parts. In the first part, we will introduce general principles of type analysis and related optimizations of dynamic or semi-dynamic values. In the second part we will present formalization of type analysis, which is directly used by our algorithm we have implemented and that will be described in chapter 5. In the third part we will discuss advanced techniques that can be used for optimization. Especially we will introduce type specializations, which seem to be a very strong technique for optimization of PHP programs.

Typically, type analysis is used in compilers of statically typed functional languages to assign types to value bindings in order to produce succinct code by omitting type annotations. Its goal is to find a mapping of types to values, preferring the most general assignments. In our case the problem is different as a dynamic language program will work without knowing types at compile-time per se. In an ideal case, we would benefit from any limitation of set of types of a variable or other typed entity that will be possible to get at compile-time. This is not exactly as in reality but illustrates what we will try to achieve. We will formalize our goal further.

Without type analysis, the compiler is limited in making optimizations only using entities known at compile-time – classes, functions, constants and literals. While this may seem as a contradiction with the language's dynamicity, in most cases it can be proven that at a specific call site in the program such entity has one and only definition possible

4.1 Current Research

First type inference algorithm was created for simply typed lambda calculus by Haskell Curry and Robert Feys [**35**]. It was later proved by Hindley [**36**] that this algorithm always infers the most general type of the expression. This result is particularly important since typed lambda calculi are among the simplest formalizations of computation with types. Notably, simple typed lambda calculus predates the existence of computers and programming languages.

Originally, type inference was used in the domain of functional languages, where it became invaluable part of language specification, relieving programmers from specifying types and often allowing the compiler to automatically produce polymorphic functions. With propagation of functional features into static languages, type inference algorithms were introduced into mainstream imperative languages as well. For example, in *C# 4.0* lambda function construct, types of arguments are inferred from delegate type of a variable or an argument the lambda function is being assigned to.

4.1.1 Cartesian Product Algorithm

Cartesian Product Algorithm [**37**] is a type inference algorithm for Self programming language [**18**], that deals with the complex hierarchy of types that is present in the language. It is often cited as an advanced algorithm that can quickly and efficiently deal with system of prototypes in Self.

Since PHP does not have dynamic system of classes, we will not use the important results of this algorithm, albeit we know of it and may have found and inspiration in it.

4.1.2 DRuby

In [**38**] authors are describing type inference for *Ruby* dialect called *DRuby*, that combines static typing with dynamic typing. Ruby is in some aspects similar to PHP and therefore results of this research are beneficial for us.

Generally, algorithm discussed in the paper is constraint-based and visits each statement in the program once, while trying to maximize the results. It is performed on an intermediate representation called *RIL – Ruby intermediate language*. The algorithm itself is intended for finding type-based errors in the language and models the language very well, helped by user type annotations.

4.1.3 PHC

As mentioned earlier, *PHC*, as presented in [**8**], is a compiler of *PHP* language that performs several static program analyses in order to devise a fast executable. As we have stated earlier (see section 2.1) the compiler does not currently model several important aspects of PHP such as classes or *EVAL* statement, nor is any prospect of dealing with these constructs presented in the work itself.

In the text, authors propose several approaches to type inference, which we have described earlier. These analyses are not tested and only optimizations performed by the compiler are constant-propagation and dead-code elimination that are not covered in this thesis and are not related to types and type inference. However, it is important to note that due to time constraints we haven't inspected nor searched for subsequent research papers regarding PHC.

4.1.4 JavaScript

JavaScript [**39**] is a most popular client-side scripting language that enables web pages to alter their HTML source dynamically based on user's input. It is a dynamic language with completely dynamic type system with prototyping and duck type checking. This means that the type system is much more complicated than that of PHP.

In [**40**], the authors discuss fairly complex type inference algorithm for JavaScript that makes use of lattice formalization, especially that of prototypes, which are most important in JavaScript, which is typically used in object-oriented way.

4.2 General Principles

As noted earlier in section 1.4, *Phalanger* and the whole *.NET* environment have advantage over *C*-based implementation in the runtime type information (called Reflections). *Phalanger* exploits this principle by strongly typing library functions that are implemented in *C#*, thus providing starting point for type inference, which will have many times more typed variables than in environment where this is not possible.

Specifically, *PHC* and *HPHP* know types only from literals and operator semantics. This will allow us to perform much more aggressive optimizations while keeping them conservative and thus not changing the program itself.

In the following sections, we will presume that control-flow graph as described in Chapter 3 was already built on the source code and in cases where difference between global and local code was discussed, we will also presume that global control flow analysis is performed.

4.2.1 Temporal typing

The key difference between static and dynamic language for type inference is that instead of assigning a variable with type near the beginning of its scope and then proving that such typing will work we will be building knowledge about the type of a variable at each point of its scope (we call this temporal typing). The reason for this is obvious – in dynamic language context, typing can change during the program's execution. Presence of dynamic operations results in global typing, in which most variables are typed dynamically.

```
1:      $a = 1;          // $a : int
2:      $b = foo($a);    // foo(int) : ‘a
3:      $a = “text”;     // $a : string
4:      bar($a, $b);     // bar(string, ‘a) : void
```

Snippet 10 - Temporally typed variables.

In Snippet 10 we show a part of a program in which a variable changes its type, but is always containing value of a single known type. Variable "*$a*" is assigned with an *integer* value on line 1 and then a *string* value on line 3. These values are then passed to functions *foo* and bar. Had we used the classical type inference, we wouldn't be able to infer a type of the variable "*$a*" in any way and hence functions *foo* and *bar* will be passed fully dynamic value.

Notion of temporal typing will help us to optimize the code even if a variable is used with several different types. While in local scopes this does not happen very often, global variables will certainly benefit of this, especially in applications with large portion of global code - commonly used by PHP programmers. Additionally, we will describe that temporal typing will make optimization of indirectly accessed places, such as array items and object properties, possible.

SSA Form

Obvious problem of temporal typing is a fairly large amount of memory consumed for storing lattice values for each point of code. This can be solved by using frequently used *Static Single Assignment Form* [**41**], in which the temporality is inherent since each variable is assigned only once and allocation of local variables is then reconstructed by traversing the control flow graph. This has several advantages such as lower memory requirements and inherent flow sensitivity of otherwise flow insensitive analyses.

However, in context of dynamic languages the *SSA* form is usable directly only when no dynamic operation is present in code that uses the variable. In case such operation can occur, *SSA* form would need to be changed so that each virtual variable created by transformation to *SSA* is bound with its original name. A dynamic operation can then access only variables that would be last of such name, older variables being unusable.

When the code generation phase is reached, all valid *SSA* variables need to be flushed into a structure in which they would be accessible by their name before the dynamic operation occurs. This ensures that this operation has access-by-name to the actual state of the local scope.

4.2.2 Variable Scopes

In *PHP*, we need to differentiate between variables based on their scope – global variables and local variables. Temporal typing of local variables is usually straightforward because they are only accessed by the function itself[9]. Consequently, if a variable or its alias[10] is not passed to another function, its value is not changed by other function or code. Lifetime of a local variable ends when the control is returned to function's caller, with exception of it being aliased during the execution and the alias was not stored in a globally accessible variable, return value or an argument variable passed by reference.

Global variables are more complicated since their lifetime lasts for whole program execution and therefore any alteration of it must be taken into account. In PHP, access to global variables from a local scope must be declared as shown in Snippet 11.

Interesting property of PHP is that the global variable usage declaration is flow-sensitive. The canonical way of accessing the global variable is through super-global array "*__GLOBALS*". To transform the program to canon, we would need to remove the flow-sensitive global usage declaration by declaring a local Boolean variable that would decide whether we will access global or local variable.

```
1:      $a = 1;                 // global $a is integer
2:      function foo($x) {
3:              $a = 0;         // local $a is integer
4:              if ($x) global $a;      // $a now refers to a global value
5:              return $a;      // returns always integer
6:      }
7:      echo foo(true);         // prints "1"
8:      echo foo(false);        // prints "0"
```

Snippet 11 - Declaration of Global Variable Access

Global variables that are accessed from other places than from the script that is using them have almost always a single type and are set only once in the program and only read afterwards. Such variables store context object, cache handling objects etc. Therefore, keeping the type information of global variables is important, since we will then be able to remove the dynamic dispatch from call sites that are targeting objects in global variables.

4.2.3 Union Types

Since functions[11] in dynamic languages tend to have more than one possible return type, we need to represent such values at runtime – otherwise we would need to fall back into a generic value type almost every time. We call such runtime representation of multi-type value an *Union* type.

Such types have to be handled automatically by a compiler, which can use them to optimize access operations. Because a value with Union type is more specialized than an arbitrary value all

[9] There are few exceptions to this – extension functions can access the local scope of the caller and inclusions in local scope can access local variables of including function – both will discussed later in this section.

[10] By alias we mean other variables containing PHP-style reference pointing on the same value. See subsection 4.4.4 for a description of PHP references.

[11] For brevity, by functions we mean also all operators and similar operations that do not produce locally observable side-effects. We will clarify this later in the text.

operations with it are faster. On the other hand, having more than few types in such *Union* are not needed because the less certain the type is the less it is useful. The exact maximum of usable size of *Union* types varies language by language.

For example, if we had two possible types for a value, we will be doing much less runtime checks with *Union* type than if we had treated the value as arbitrary. The memory needed is almost the same, but we will increase the performance.

Particularly in PHP, having *Union* types is very important because PHP semantics include common implicit conversion between types based on variable values, e.g. instead of overflowing, an *integer* will eventually become a *double*. Consider following example with function random which returns random *integer* from full range of possible values and "*dist*" which takes a double as an argument and returns a double.

```
1:      $a = random();          // $a : integer
2:      $b = $a * 2;            // $b : <integer, double>
3:      $c = dist($b);          // $c : double
```

Snippet 12 – Union Types

In Snippet 12, it is shown that if we did not use Union types, we would need to either change semantics of the "*" operator or defer variable $b to an arbitrary return type. In the first case, we won't be able to express half of the possible values. The latter would subsequently lead to "*dist*" being processed by the algorithm with dynamic value as an argument, which will eventually lead to very slow evaluation because of required type checking. On other operators for *Union* type <*integer*, *double*> are fairly less complex and deal with only two possible values – and they are significantly faster.

4.2.4 Option Types

For efficient implementation of object properties that have at least some type information inferred, we define notion of option types that are somewhat similar to Union types and option types known from functional languages, but not exactly equivalent. These types are specialized into a specific representation of value that is most likely to occur – both primitive, object or Union type can be in this place, but they keep an option of falling back into a generic representation. Whether the specialized or generic version is to be used is decided using a Boolean value, which is member of the option type.

Reading from a property typed with an option type adds only a single test and branching operation both in the best and the worst case. Writing to such property in the best case does not add any test or branching since we know that the written value has the specific type, we only need to update the option value, which has negligible cost compared to reading the generic value. Worst case for writing does not happen very often and it needs to test whether the assigned value is of the specialized type and in this case it needs to properly set the state of the option type.

Other approach to the worst case scenario is to ignore that the value can have a specialized type of the option type and simply assign the value into the generic version, which would speed up the assign and slow the reading. However, it will then be needed to make a type check even in places where we are sure about the type of the contained value.

As such, option types are heuristic data structure that can significantly speed up access to object fields and properties while keeping the generic functionality, which is only marginally slower. Efficiency of option types depends on the accuracy of the type analysis which can, for reasonable programs, infer the probable, or in static context precise, type of the property.

Option types can also be used for global variables in case we cannot prove that a single type of value is assigned to global variable, usually thanks to *EVAL* statement performed somewhere in the program. We can then heuristically presume that the value will retain the inferred type even when *EVAL* does occur in the program (which is obviously valid heuristic for reasonably designed applications).

4.2.5 Dynamic Operations

The main obstacle to doing type analysis in a dynamic language is presence of *EVAL* statement, script dependency statement, indirect variable access, indirect calls. If an *EVAL* operation cannot be ruled out by preceding analyses, it makes usage of most information collected by type analysis difficult. This is because all what we are trying to achieve with the analysis is limiting of value semantics (untyped variables) to their small subset (typed variables), which would allow us to subsequently generate faster code. Since *EVAL* operation can change almost everything in the program, we need to be very careful with inferring types within *EVAL* statement's reach.

Thankfully, PHP developers tend not to overuse *EVAL* and the like because it is slow and makes code harder to understand and very difficult to debug.

Eval Operation

In a general case, after *EVAL* operation occurs we need to drop all typing inferred prior to this operation that *EVAL'ed* code could have changed directly or by semantically observable side-effects – in most cases this is everything accessible by the calling a function. We can start inferring types again after the possibility of *EVAL* execution is ruled out, presuming that all variables are completely dynamic. There are two improvements to this.

First improvement uses a heuristic that *EVAL* code will use variables in the same manner as they were typed previously. To express this in variable typing, we need to introduce new predicate that expresses the uncertainty of the heuristic, which will enable consumers of inferred typing to effectively deal with the situation.

Second improvement involves passing the inferred typing prior to *EVAL* statement. This allows compiler processing *EVAL* statement to have better starting point for its own type analysis. We will not discuss this further, since it is out of scope of this work because *EVAL* operations are not used frequently in our host language - *PHP*. The only note we will make is that caching of such compiled *EVALs* is needed since compiler runs very slowly and naturally the inferred type information has to be part of cache key in such case. Checking equality of such structure can be very time-consuming.

Same as other dynamic language features, *PHP* is very specific in what *EVAL* can and cannot break. The basis of this is previously mentioned monotonic property of declarations. *EVAL* cannot alter code of previously defined functions and objects but it can use those definitions and supply them with values that do not match inferred typing. There are following consequences of possibly executed *EVAL* statement in PHP:

1. Global variables have to lose all typing before *EVAL*'s execution. This is obvious since *EVAL* statement can assign any value into them. Note that *EVAL* code can access global variables even if executed in a function or a method scope. The same applies to local variables of a function or method if they contain *EVAL*.
2. Declarations preceding the *EVAL* statement can be called by *EVAL'ed* code with a different argument typing than type inference concluded. However calls to such functions and classes can be early bound because *EVAL* cannot break PHP's declaration monotonicity – if it does re-declare the entity, a runtime error is raised.
3. Unconditional declarations succeeding execution of *EVAL* can be treated normally – early bound with the argument type inference. If *EVAL* had declared entity of the same name beforehand, a runtime error will be raised when passing the declaration, so the logic won't be impacted.

Other Operations

Indirect call operation can have a subset of effect of an *EVAL* operation, but in the worst case it can have the same effect. That occurs if any of functions defined at the point of indirect call execution contains *EVAL* operation. In other cases, indirect call can only call all currently defined functions, which is a strong capability, but the side effects performed are still bounded.

Indirect variable access can only alter state of a local or a global variable. This can have very severe effects on type information collected about the variables. Global variables can be always accessed through the super global "__GLOBALS" variable that is available in all scopes.

Dynamic script inclusion is equivalent to *EVAL*, since it can include arbitrary script from the hard drive, which is present in here during the execution of the program. Inclusion taking place in a function can access local variables of that function and inclusion in global code can access global variables.

4.3 Formalization

In this section we will briefly formalize our approach to type analysis. We won't be covering every detail of previously described principles for sake of brevity of this text. There are some aspects, notably PHP arrays, which could be treated in more detail, but we deem it out of scope of the work.

The type analysis is a data-flow-based analysis, which is performed on the edges of the data flow graph and which is used to assign types to functions, object properties and both local and global variables. In order to implement *Union* types as described in section 4.2.3, we need a structure that would describe a set of possible types in a variable. Generally in compiler theory, lattices over "supertype" relation are chosen as such structure, because of having global infimum (i.e. no value assigned) and for each two values, we can find a supremum (i.e. all possible value can be assigned to the variable).

Using such lattice, behavior of program's operation is modeled using transition (or transfer) functions that always result in lattice member superior or equal to supplied arguments. Possible values of variables are then analyzed by using transition functions along the edges of the control-flow graph until a fix point is reached.

4.3.1 Type Lattices

In both static and dynamic language compilers, complete lattices [**42**] represent the hierarchy of nominative types and their values defined by the language. For dynamic languages with completely dynamic type systems, such lattices are very complicated because they need to model all aspects of the dynamic type systems, such as primitive values and all possible object values [**40**]. In case of PHP, the problem is somewhere in the middle – PHP has partially a static nominative type system and programmers are used to use it consistently (see 1.2.2).

The type analysis in a dynamic language targets to find a supremum of all possible values of a variable in the lattice. As stated in section 4.2.3, we target to use Union types at runtime. Hence, we model a value of a variable in PHP using a lattice $Value$ that is defined in the following way, similarly to JavaScript value lattice as defined in [**40**], which will represent multiple possible types that a variable can be assigned with during the execution:

$$Value = Null \times Boolean \times Number \times String \times Array \times Object$$

This composite lattice uses single-type lattices as its elements and single-type lattice relations are combined into a composite relation that forms a lattice on the set. The composite relation is defined in the following way:

$$\forall x = (x_1, x_2, x_3, x_4, x_5, x_6) \in Value \forall y = (y_1, y_2, y_3, y_4, y_5, y_6) \in Value:$$

$$x \prec_{Value} y \Leftrightarrow x_1 \prec_{Null} y_2 \vee x_2 \prec_{Boolean} y_2 \vee x_3 \prec_{Number} y_3 \vee x_4 \prec_{String} y_4 \vee x_5 \prec_{Array} y_5 \vee x_6 \prec_{Object} y_6$$

Such relation forms a lattice on the $Value$ set, because if we have two arbitrary values $x, y \in Value$ we are able to find both supremum and infimum by using the element relation supremums and infimums on elements of x and y. Such definition of supremum and infimum on $Value$ is indeed defined for all $x, y \in Value$, thus giving us a complete lattice.

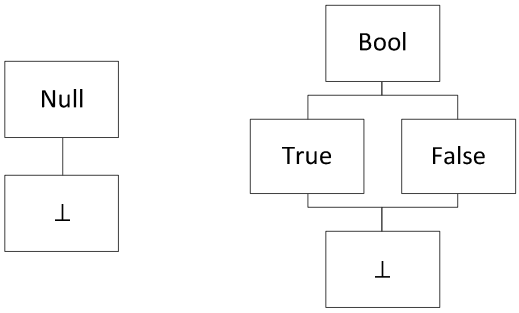


Figure 23 - Diagrams of Null and Boolean lattices

The Lattice $Null$ describes a possible empty value assigned to the variable and therefore it contains only top and bottom items. Lattice $Boolean$ describes a Boolean value and apart from top and bottom, it contains TRUE and FALSE values representing Boolean constants. Both lattices are shown in Figure 23. Constant value item of the lattice are important because they will help us to perform the constant propagation alongside type analysis.

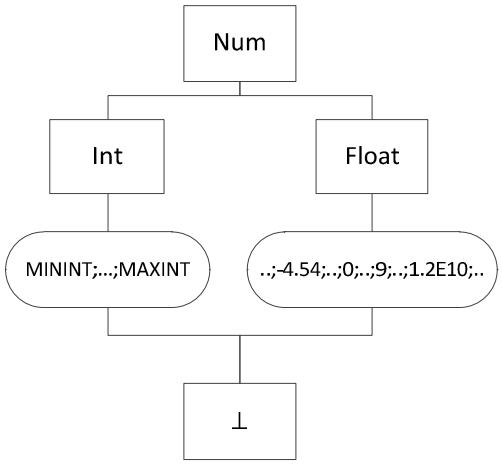


Figure 24 - Diagram of Number lattice

The lattice $Number$ represents both integer and floating point values, as shown in Figure 24. This unification of representation is consequence of our conception of *Union* types with this lattice being the most important example.

Arithmetical operations in PHP have the property of overflown integer values becoming floating point values. Because of that the $Number$ *Union* type will be needed in most cases as we are not able to prove whether an integer value is small enough not to overflow. The $Number$ lattice additionally represents all possible values of Integer and Float as constants, including differentiation between negative and positive zero present in IEEE representation of 64-bit floating point values. Whole numbers are present separately for *Integer* and *Float* as their semantics differ.

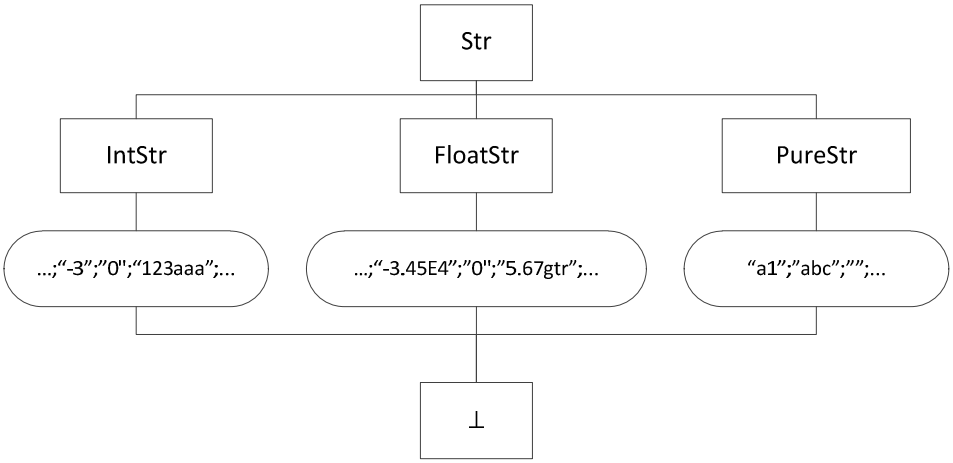


Figure 25 - Diagram of String lattice

The lattice $String$, shown in Figure 25, represents string values and is more complicated, since string values are implicitly converted to *Integer* and *Float* types. This conversion is not only using string representation, but even when arbitrary string is appended to the string representing a number it also converts into a number that is in the beginning of the string. We represent this using *IntStr* and *FloatStr* item and subsequent constants. Other string values are represented by *PureStr* item and underlying constants. This specification of lattice omits transition details such as decimal point character or exponent character or the whole convertibility of string to Boolean. This would however

make the lattice too complicated and so we have chosen not to include every possible detail of the string lattice specification.

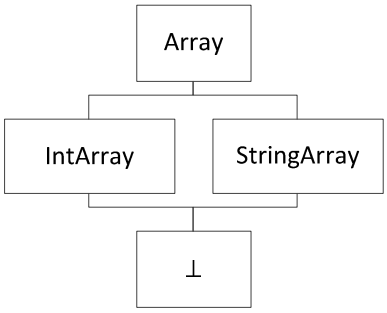


Figure 26 - Diagram of Array lattice

The $Array$ lattice can be represented in many ways, since there are many representations of PHP array that are more efficient than the generic array, which is a hash table with two different types of keys and arbitrary values. For brevity, we provide only a simple lattice representation that differentiates between different types on indices as shown in Figure 26.

It is possible to model the lattice for arrays in greater detail, which can be still useful for optimization. For example, we can model linear arrays that contain indices, which are almost without gaps. Such PHP array would be possible to be represented by an array in traditional sense, thus removing the need to use a hash table.

Another example is representation of types of array values. Since arrays usually contain only one type of values, we could create lattice that would express it. In many algorithms that use arrays, we will be able to have type information for array values and we will be able to speed up both reading and writing to the array.

The last example would be modeling of structures that are sometimes saved in the array using constant indices. Values for each index usually have a single type, which can be inferred. Using this approach we would be able to express large database representations and represent them by object tree, completely removing the need for arrays as such. However, this analysis would be very complicated.

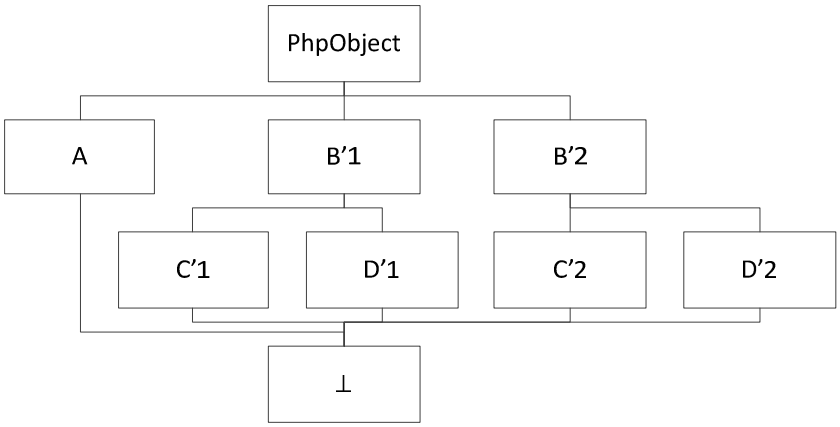


Figure 27 - Example of Object lattice

The last component of the $Value$ lattice is a hierarchy of object types that is defined by the program. The definition of this lattice has to take into account conditional declarations of classes, thus creating different branch for each version of the same class and also duplicating the derived classes for each version of their base class. Example of such lattice is shown in Figure 27.

To express a lattice value $V \in Value$, we will use the vector notation. For example, if V is an *integer* or a *string*, it would be represented by a vector $(\bot, \bot, \mathrm{Int}, \mathrm{NonNumString}, \bot, \bot)$, where *bottom* signs $\bot$ represent the (current) certainty, that such value would not occur at runtime.

Additionally, we define set of functions, which will get us respective part of the lattice value, defined as follows:

$$(Null(X) \to a) \forall X = (a, b, c, d, e, f) \in Value$$

$$(Boolean(X) \to b) \forall X = (a, b, c, d, e, f) \in Value$$

$$(Number(X) \to c) \forall X = (a, b, c, d, e, f) \in Value$$

$$(String(X) \to d) \forall X = (a, b, c, d, e, f) \in Value$$

$$(Array(X) \to e) \forall X = (a, b, c, d, e, f) \in Value$$

$$(Object(X) \to f) \forall X = (a, b, c, d, e, f) \in Value$$

We will use these function in transition function rules described in the next subsection. Then, we will use operator $\prec$ as a relation in each component of the $Value$ lattice.

4.3.2 Transition Functions

To model the language, a set of transition (or transfer) functions has to be defined, which will be used on the edges of control-flow graph in order to attain typing information of all variables. More precisely, for each assignment in the *Static Single Assignment* (SSA) form we will perform transition functions on all sub-expressions of the assigned value starting at leafs and going to the root of assigned expression. In place of variables, we will use already obtained lattice values for those variables. Literals and constant have their values predefined. Other starting points for transition functions are library functions that are often strongly typed and therefore we are able to determine the lattice type information for their return value.

This assigning of types is performed repeatedly until a fixed point is reached. From definitions of transfer functions for each language construct, function or operator, which we presume to monotonous, i.e. for parameters that have more general or equal lattice value than previous parameters returning more general or equal value than previous return value. Therefore, since lattice has limited depth and analysis is performed on limited source code, there exists a fix point and the algorithm will eventually in limited time converge to it.

We will provide only few examples of transition functions for built-in operators, while other language constructs will be discussed in the next section in less detail informally. We present an addition and concatenation operators that represent the most complicated transfer function for PHP.

Addition Operator

First example for transition function is addition operator that adds two values. In PHP, addition operator must express so called type juggling, i.e. implicit conversions by which variables change their real type without programmer explicitly controlling it. For instance, as mentioned earlier, when two large integers would overflow, they are converted to floating point values and added with floating point semantics.

To define a transition function $Add: Value \times Value \to Value$, we will first define a function that represents semantics of number addition, i.e. inference function $AddNumbers: Number \times Number \to Number$. This function is for each $X, Y \in Number$ being operands the function defined by following rules:

1) If X and Y are constant *integer* values, i.e. $X \prec Int$ and $Y \prec Int$, $AddNumbers(X, Y)$ is equal to a single constant value that can be either integer or float, i.e. $\exists! z: (z \prec Int \vee z \prec Float) \wedge z = AddNumbers(X, Y)$. The actual type depends on the fact whether the value has overflown or not.
2) If X and Y are constant *float* values, i.e. $X \prec Float$ and $Y \prec Float$, $AddNumbers(X, Y)$ is always a float constant value, i.e. $\exists! z: z \prec Float \wedge z = AddNumbers(X, Y)$.
3) If X is a constant *float* and Y is a constant *integer*, i.e. $X \prec Float$ and $Y \prec Int$, $AddNumbers(X, Y)$ is always a float constant value, i.e. $\exists! z: z \prec Float \wedge z = AddNumbers(X, Y)$. This also applies for the symmetrical case.
4) For arbitrary *integer* values, i.e. $X = Int$ and $Y \preccurlyeq Int$, the result is always generic: $AddNumbers(X, Y) = Num$. This rule also represents case when there is a constant on one side and also applies to the symmetrical case.
5) For arbitrary *float* values, i.e. $X = Float$ and $Y \preccurlyeq Float$, the result is always float: $AddNumbers(X, Y) = Float$. This rule also represents case when there is a constant on one side and also applies to the symmetrical case.
6) For left hand operand being a *float* and right hand operand being an *integer*, i.e. $X \preccurlyeq Float$ and $Y \preccurlyeq Int$ and not both constants, i.e. $\neg(X \prec Float \wedge Y \prec Int)$, the result is always a *float*, i.e. $AddNumbers(X, Y) = Float$. The same applies for the symmetrical case.
7) For left hand operand being Num and right hand operand being an arbitrary value, i.e. $X = Num$ and $Y \preccurlyeq Num$ the result is always Num, i.e. $AddNumbers(X, Y) = Num$. The same applies for the symmetrical case.
8) Operation is not defined for any operand being $\bot_{Number}$ (this is the same for all operators).

Values of other types - $Null$, $Boolean$, $String$, $Object$ are first converted to $Number$ using a conversion function and then applied to $AddNumbers$ function. Conversion from an $Array$ to a $Number$ is invalid so we do not include its definition as it leads to both compile time and runtime error if encountered. Hence, we define the function $StringToNumber: String \to Number$ that maps a member of the string lattice to a member of the values lattice. It is defined by following rules for $X \in String$:

1) $X \prec IntStr \implies \exists! Y \prec Int \wedge StringToNumber(X) = Y$
2) $X \prec FloatStr \implies \exists! Y \prec Float \wedge StringToNumber(X) = Y$
3) $X = IntStr \implies StringToNumber(X) = Int$
4) $X = FloatStr \implies StringToNumber(X) = Float$
5) $X \preccurlyeq PureStr \implies StringToNumber(X) = 0 \prec Int$

6) $X = Str \Longrightarrow StringToNumber(X) = Num$
7) $X = \bot_{String} \Longrightarrow StringToNumber(X) = \bot_{Number}$

Then, we define the function $BoolToNumber: Bool \rightarrow Number$ that maps a member of $Boolean$ lattice for $X \in Boolean$ in the following way:

1) $X = True \Longrightarrow StringToNumber(X) = 1 \prec Int$
2) $X = False \Longrightarrow StringToNumber(X) = 0 \prec Int$
3) $X = Bool \Longrightarrow StringToNumber(X) = Int$
4) $X = \bot_{Boolean} \Longrightarrow StringToNumber(X) = \bot_{Number}$

Consequently, the function $NullToNumber: Null \rightarrow Number$ is defined by following rules:

1) $X = Null \Longrightarrow StringToNumber(X) = 0$
2) $X = \bot_{Null} \Longrightarrow StringToNumber(X) = \bot_{Number}$

For brevity, we will not include a definition of the function that converts $Object$ to $Number$, $ObjectToNumber: Object \rightarrow Number$ since we have not described objects in needed depth.

Having all needed functions, we can define the transition function $Add: Value \times Value \rightarrow Value$ as follows:

$$X_S := \sup_{\text{Number}} \begin{Bmatrix} Number(X), StringToNumber\big(String(X)\big), BoolToNumber\big(Bool(X)\big), \\ NullToNumber\big(Null(X)\big), ObjectToNumber\big(Object(X)\big) \end{Bmatrix}$$

$$Y_S := \sup_{\text{Number}} \begin{Bmatrix} Number(Y), StringToNumber\big(String(Y)\big), BoolToNumber\big(Bool(Y)\big), \\ NullToNumber\big(Null(Y)\big), ObjectToNumber\big(Object(Y)\big) \end{Bmatrix}$$

$$Add(X, Y) = \{\bot_{Null}, \bot_{Boolean}, AddNumbers(X_S, Y_S), \bot_{String}, \bot_{Array}, \bot_{Object}\}$$

Supremum operation infer the most general conversion to number that is contained in X and Y. As we can see, the addition operator always returns a number, which is important as it will allow type analysis to frequently obtain very narrow values of $Value$ lattice in terms of included types. We can demonstrate this rule on several simple examples:

$$Add(\{\bot, \bot, 3, \bot, \bot, \bot\}, \{\bot, \bot, 67, \bot, \bot, \bot\}) = \{\bot, \bot, 70, \bot, \bot, \bot\}$$

$$Add(\{\bot, \bot, 56.3, \bot, \bot, \bot\}, \{\bot, \bot, \bot, "78a", \bot, \bot\}) = \{\bot, \bot, 134.3, \bot, \bot, \bot\}$$

$$Add(\{\bot, True, \bot, \bot, \bot, \bot\}, \{\bot, \bot, \bot, PureStr, \bot, \bot\}) = \{\bot, \bot, 1, \bot, \bot, \bot\}$$

$$Add(\{\bot, False, Int, \bot, \bot, \bot\}, \{Null, \bot, \bot, Str, \bot, \bot\}) = \{\bot, \bot, Num, \bot, \bot, \bot\}$$

Addition operator will most often give us Num, $Float$ and constants since we are not considering any king of range analysis that would help us limit Int values so that we will be able to get Int as a result of an addition.

Concatenation

Concatenation operator transfer function is defined similarly to that of the addition operator with the exception that it converts all values to strings before they are passed to supremum and then it goes through specific set of rules. The important feature of concatenation is the fact that it handles

string convertibility of both floats and integers. If a convertible string is concatenated with any other string on the right hand side, the resulting string remains to be convertible. We will not describe all details and in very obscure cases we will simply let the result become generic Str as expressing all possible combinations of concatenated string will be very lengthy. However, for an illustration this will suffice.

We define a transition function of concatenate operator as a mapping $Concat: Value \times Value \rightarrow Value$, that is defined using an inference function $ConcatStrings: String \times String \rightarrow String$ and conversion functions $NullToString$, $BoolToString$, $NumberToString$, $ArrayToString$, $ObjectToString$ as follows:

$$X_S \coloneqq \sup_{\text{String}} \begin{Bmatrix} String(X), NumberToString\big(Number(X)\big), ArrayToString\big(Array(X)\big) \\ NullToString\big(Null(X)\big), BoolToString\big(Bool(X)\big), ObjectToString\big(Object(X)\big) \end{Bmatrix}$$

$$Y_S \coloneqq \sup_{\text{String}} \begin{Bmatrix} String(Y), NumberToString\big(Number(Y)\big), ArrayToString\big(Array(Y)\big) \\ NullToString\big(Null(XY)\big), BoolToString\big(Bool(Y)\big), ObjectToString\big(Object(Y)\big) \end{Bmatrix}$$

$$Concat(X,Y) = \{\perp_{Null}, \perp_{Boolean}, \perp_{Number}, ConcatStrings(X_S, Y_S), \perp_{Array}, \perp_{Object}\}$$

The inference function, as mentioned above, handles a concatenation of two strings, primarily based on their type, i.e. *int-convertible*, *float-convertible* and *pure string* and their respective constant values. We do not define all details such as concatenation of "." with an *int-convertible* string resulting in *a float-convertible string*. This behavior happens in reality, but we can always simplify its representation by inferring a generic *string*. Hence, we define $ConcatStrings: String \times String \rightarrow String$ by following rules for operands X and Y:

1) $X \prec IntStr \wedge Y \prec PureStr \Longrightarrow \exists! z \prec IntStr: ConcatStrings(X,Y) = z$
2) $X \prec FloatStr \wedge Y \prec PureStr \Longrightarrow \exists! z \prec FloatStr: ConcatStrings(X,Y) = z$
3) $X \prec IntStr \wedge Y \prec IntStr \Longrightarrow \exists! z \prec IntStr: ConcatStrings(X,Y) = z$
4) $X \prec IntStr \wedge Y \prec FloatStr \Longrightarrow \exists! z \prec FloatStr: ConcatStrings(X,Y) = z$
5) $X \prec FloatStr \wedge Y \prec IntStr \Longrightarrow \exists! z \prec FloatStr: ConcatStrings(X,Y) = z$
6) $X \prec FloatStr \wedge Y \prec FloatStr \Longrightarrow \exists! z \prec FloatStr: ConcatStrings(X,Y) = z$
7) $X \prec PureStr \wedge Y \prec IntStr \Longrightarrow \exists! z \prec PureStr: ConcatStrings(X,Y) = z$
8) $X \prec PureStr \wedge Y \prec FloatStr \Longrightarrow \exists! z \prec PureStr: ConcatStrings(X,Y) = z$
9) $X \prec PureStr \wedge Y \prec PureStr \Longrightarrow \exists! z \prec PureStr: ConcatStrings(X,Y) = z$
10) $X \preccurlyeq IntStr \wedge Y = PureStr \Longrightarrow ConcatStrings(X,Y) = Str$
11) $X = IntStr \wedge Y \preccurlyeq PureStr \Longrightarrow ConcatStrings(X,Y) = Str$
12) $X \preccurlyeq FloatStr \wedge Y = PureStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$
13) $X = FloatStr \wedge Y \preccurlyeq PureStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$
14) $X \preccurlyeq PureStr \wedge Y = PureStr \Longrightarrow ConcatStrings(X,Y) = PureStr$
15) $X = PureStr \wedge Y \preccurlyeq PureStr \Longrightarrow ConcatStrings(X,Y) = PureStr$
16) $X \preccurlyeq IntStr \wedge Y = IntStr \Longrightarrow ConcatStrings(X,Y) = IntStr$
17) $X = IntStr \wedge Y \preccurlyeq IntStr \Longrightarrow ConcatStrings(X,Y) = IntStr$
18) $X \preccurlyeq FloatStr \wedge Y = IntStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$
19) $X = FloatStr \wedge Y \preccurlyeq IntStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$
20) $X \preccurlyeq PureStr \wedge Y = IntStr \Longrightarrow ConcatStrings(X,Y) = PureStr$
21) $X = PureStr \wedge Y \preccurlyeq IntStr \Longrightarrow ConcatStrings(X,Y) = PureStr$
22) $X \preccurlyeq IntStr \wedge Y = FloatStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$

23) $X = IntStr \wedge Y \preccurlyeq FloatStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$
24) $X \preccurlyeq FloatStr \wedge Y = FloatStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$
25) $X = FloatStr \wedge Y \preccurlyeq FloatStr \Longrightarrow ConcatStrings(X,Y) = FloatStr$
26) $X \preccurlyeq PureStr \wedge Y = FloatStr \Longrightarrow ConcatStrings(X,Y) = PureStr$
27) $X = PureStr \wedge Y \preccurlyeq FloatStr \Longrightarrow ConcatStrings(X,Y) = PureStr$
28) $X = IntStr \wedge Y = Str \Longrightarrow ConcatStrings(X,Y) = Str$
29) $X = FloatStr \wedge Y = Str \Longrightarrow ConcatStrings(X,Y) = FloatStr$
30) $X = PureStr \wedge Y = Str \Longrightarrow ConcatStrings(X,Y) = PureStr$
31) $X = Str \wedge Y = IntStr \Longrightarrow ConcatStrings(X,Y) = Str$
32) $X = Str \wedge Y = FloatStr \Longrightarrow ConcatStrings(X,Y) = Str$
33) $X = Str \wedge Y = PureStr \Longrightarrow ConcatStrings(X,Y) = Str$
34) $X = Str \wedge Y = Str \Longrightarrow ConcatStrings(X,Y) = Str$
35) $X \preccurlyeq Str \wedge Y = \perp_{String} \Longrightarrow ConcatStrings(X,Y)\ not\ defined$
36) $X = \perp_{String} \wedge Y \preccurlyeq Str \Longrightarrow ConcatStrings(X,Y)\ not\ defined$

As we can see, even very simplified string lattice results in very complex behavior of concatenation inference function. Note that if we would express also "E", ".", "TRUE" and "FALSE", we would need also to define all string that begin with "E". This would result in a lattice with several thousand of similarly simple rules that are efficiently implemented in a programming language. Modeling the lattice to such details does not have many benefits as the specialized values in strings are of concern when at least one of strings that are concatenated together is a constant and then the string is used in an arithmetical operation. This combination does happen often and so in our opinion this detail of string lattice is sufficient.

We then continue with definition of string-conversion functions, beginning with $NumberToString: Number \rightarrow String$, which is defined by following set of rules:

1) $X \prec Int \Longrightarrow \exists!\, z \prec IntStr: NumberToString(X,Y) = z$
2) $X \prec Float \Longrightarrow \exists!\, z \prec FloatStr: NumberToString(X,Y) = z$
3) $X = Int \Longrightarrow NumberToString(X,Y) = IntStr$
4) $X = Float \Longrightarrow NumberToString(X,Y) = FloatStr$
5) $X = Number \Longrightarrow NumberToString(X,Y) = Str$
6) $X = \perp_{Number} \Longrightarrow NumberToString(X,Y) = \perp_{String}$

Conversion functions $NullToString$ and $BoolToString$ are trivial as they directly convert constants and generic value becomes $PureStr$. The $ArrayToString$ conversion function results always in $PureStr$ and $ObjectToString$ always in Str.

We will now demonstrate how concatenation operator works. Let us consider following examples:

$$Concat(\{\perp, \perp, \perp, "a", \perp, \perp\}, \{\perp, \perp, 10, \perp, \perp, \perp\}) = \{\perp, \perp, \perp, "a10", \perp, \perp\}$$

$$Concat(\{\perp, \perp, Float, \perp, \perp, \perp\}, \{\perp, \perp, \perp, Str, \perp, \perp\}) = \{\perp, \perp, \perp, FloatStr, \perp, \perp\}$$

$$Concat(\{\perp, False, Int, \perp, \perp, \perp\}, \{\perp, \perp, \perp, Str, \perp, \perp\}) = \{\perp, \perp, \perp, Str, \perp, \perp\}$$

$$Concat(\{\perp, Bool, \perp, \perp, \perp, \perp\}, \{\perp, \perp, Number, Str, \perp, \perp\}) = \{\perp, \perp, \perp, PureStr, \perp, \perp\}$$

$$Concat(\{\perp, \perp, 10, \perp, \perp, \perp\}, \{\perp, \perp, 10.1, \perp, \perp, \perp\}) = \{\perp, \perp, \perp, "1010.1", \perp, \perp\}$$

Arrays

In our representation of $Value$ lattice we differentiate between two specialized types of arrays based on their indices. While $Array$ lattice can be widely expanded, we will demonstrate how the simplified version can be used to provide information about the array variable.

```
1:      $a = “g”;
2:      $x[0] = 5;
2:      $x[5] = 8;
3:      $x[$a] = 4;
```

Snippet 13 - Assigning into an array

Only way a lattice value describing a variable can contain $Array$ lattice value other than $\perp$ is through array item assignment expression as seen on lines 2-4 in Snippet 13. In such case we describe the new lattice value of a variable using operator $X_{new} = ArraySet(X_{old}, T_{index}, T_{value})$. In this case we cannot benefit from SSA form as we are changing internal state of the variable type. We define the function in following way:

$$ArraySet(X_{old}, T_{index}, T_{value}) = \sup\{X_{old}, \{\perp, \perp, \perp, \perp, ArrayType(T_{index}), \perp\}\}$$

We can see that $ArrayType$ function is not dependent on the value, which is natural in our representation of $Array$ lattice. This function is defined as $IntArray$ for values that are purely $Number$, i.e. have other component equal to $\perp$, and similarly as $StringArray$ for values that are purely $String$. Otherwise, this function returns $Array$, which represents generic implementation of array.

4.4 Advanced Type Analysis

PHP contains many semantic and syntactic features that make it very complex for type analysis and which have a very weak performance. Since these features are typically used often by developers, we will look at them in this section and propose solutions to problem they pose for type analysis, which did not solve these problems so far.

We will not include detailed description or algorithms since these techniques are very complicated and developing working algorithms that would solve these problems is out of scope of this thesis. However, since we are analyzing these language features, we present brief solutions, without any formal description or proofs.

4.4.1 Functions

We have already described how calls to functions can be made static instead of dynamic, i.e. knowing exact definition of the function that is valid at the call site, if possible. This is targeted to optimize the call itself.

Analysis of variable types, in which we assign each variable an item from $Value$ lattice, gives us more information that we had not used yet. First, we are able to recognize when a function is called using constant argument or arguments. Second and more important is the fact is that for many call sites the function will be called with limited set of argument types.

First case can be exploited by expansion of the call site using function's definition and running the analysis and subsequent optimizations on the expanded part of the AST. This can give interesting result in some cases as seen in Snippet 14. If we expand function syntax tree in the call site on line 8 and reduce it using partial evaluation, we will remove both the function call and string comparison test.

```
1:      function foo($x,$y,$type) {
2:              if ($type == "jtsk")
3:                      return "JSTK($x, $y)";
4:              else
5:                      return "WGS84($x, $y)";
6:      }
7:
8:      foo($data->x, $data->y, "jtsk");
```

Snippet 14 - Function call with constant argument

More important principle is when we know at least some typing values of arguments. The type analysis as specified earlier in this chapter, if performed on a definition of a function for which doesn't have any typing of variables established yet, will begin without any knowledge of argument types and thus, in many cases, will not be able to devise much useful type information for local variables.

We will define for each reasonable typing of arguments a specialized version of the function, called a function specialization. Identifying of reasonable typing is a matter of heuristic, with conservative one being that all arguments can be represented by more specialized type than arbitrary dynamic value. As shown in Snippet 15, we are in many cases able to determine types of arguments only from context.

```
1:      function foo($x,$y,$type) {
2:              if ($type == "jtsk")
3:                      return "JSTK($x, $y)";
4:              else
5:                      return "WGS84($x, $y)";
6:      }
7:
8:      $x = $data->x * $scale_x + $offset_x;
9:      $y = $data->y * $scale_y + $offset_y;
9:      foo($x, $y, "jtsk");
```

Snippet 15 - Function with convenient specialization

Variables "$x" and "$y" are computed from the data, probably originating in a database. From the multiplication and addition, we are able to determine, that these variables will have their lattice value equal to $(\bot, \bot, Number, \bot, \bot, \bot)$. This is a strong result, since Number can be represented by a .NET's *ValueType*, which is significantly faster than arbitrary value, which is boxed in an object.

Hence, we would create a specialization of the function, which will get vector $(Number, Number, String)$ as its arguments. This specialization will be called statically with specialized arguments and it will have all dynamic behavior removed in its implementation. We

estimate that in this particular case, the whole snippet will be ten times faster than the original fully dynamic code.

By all means, number of function specializations must be kept small, so heuristics limiting the number of specializations have to be in place in real world implementation. Moreover, it is important to be aware of the fact, that if the compiled program will be used as a library, we have to include the original fully dynamic version of the function. However, in a typical program, each function will be used in most cases with very specific set of types and mostly the number of function specializations will be one.

4.4.2 Classes

Classes are frequently used as a main data structure in PHP programs. The $Value$ lattice captures them in variable type analysis. The typing is propagated through the program, starting in object constructors in case we are able to determine which definition is valid at the time (this is similar to functions). When we encounter an assign to object's field, we update the lattice value with a supremum of assigned value and original value. It is important to note, that this analysis is global and not local as other type analyses discussed in here and therefore will frequently obtain worse results.

Since in many operations and library functions the side effects made to the object are not observable, we would in most cases need to let the object to be valid even when properties are assigned with an arbitrary value. We will use previously described option types if we infer usable type information for class property.

Type analysis of methods is performed similarly as analysis of functions with the difference that a method receives reference to object on which it is invoked as the first argument. Again, method specialization can be used for different typing of arguments.

4.4.3 Arrays

As noted before, an array is a main data structure in PHP and despite this fact its implementation is very inefficient. This is implicated by the fact that a PHP array is in fact a hash table. Since many algorithms use PHP arrays as true arrays (i.e. with integer indices that do not have many gaps), it is viable to optimize arrays in these situations.

Above, we have presented a lattice representing a PHP array and we have noted that this lattice is simplified. This lattice is the definition which will have to be altered, along with definition of transition function for array item access. We would introduce a set of types that would use part of Value lattice for values in the array. The whole lattice value would be "lower" than integer-indexed array.

The caveat is, that for deciding whether an array would be "without gaps" which we used in the vague description above, we would need a knowledge that is not provided by type analysis, since it involves also a knowledge about range of values that would be in a variable, not only a type. Such analysis is called range analysis and involves searching for patterns that would help the analyzer to infer a minimum and a maximum value that can occur in a variable. While this is obviously complicated, most algorithms using arrays use a for-cycle that uses a control variable starting on 0 or 1 and going to a specified bound by incrementing the variable by 1, assigning the array with a value for each value of a control variable. In this case, we can use a typed resizable array instead of hash table, increasing the performance by a tenfold.

We can go further and include recursive types to a certain degree, since two and more-dimensional arrays are also frequently used. In our opinion this area is very interesting because as we have shown in section 2.1.10, array performance cannot be improved much without further analysis.

4.4.4 References

PHP's references are a very specific semantic feature, which is not present in other languages. We have already briefly described reference semantics in section 1.2.2. To summarize it, references in PHP have a dynamic behavior and are part of the type system. While a reference is created explicitly by using "&" operator, it is dereferenced implicitly. Because of that, it can be better described as a shared value. What makes references hard to manage is the fact that referenced (shared) value can be in any place where a value can be. That means that it is possible to have an array item, object property and a local variable sharing a value, i.e. when a change is made to it, values in all these places change at the same time.

These properties of PHP references are hard to analyze, because there is no syntactical difference between using a referenced and non-referenced value, i.e. we cannot be sure whether a change to variable will not have a side-effect in another part of program. Because of that all analyses we have previously described have to face the consequences of references.

There are two approaches – first is avoidance method and second is alias analysis. Avoidance is not entirely simple, but in general, if a reference is not explicitly passed outside of function (through an argument, return value or global variable) we can be sure that it would not hinder analysis of PHP programs. Second, alias analysis is similar to alias analysis in other languages, albeit in PHP it is needed for correctness of other analyses instead of low-level optimizations.

4.5 Summary

We have demonstrated that even in a dynamic language such as *PHP*, we are able to devise formalization of language's type system on which type analysis can be performed with results that can be used for many optimizations that remove the dynamic behavior from runtime.

We have additionally shown several advanced tasks for type analysis that would eventually, without greater effort, provide information that would help to perform heuristic optimizations that would improve performance of neatly implemented program several times.

5 Prototype Implementation

When we began implementing the theoretical concepts we have described in previous chapters, we needed to take into account the fact that *Phalanger* is implemented in tightly coupled manner. This is especially distinctive in *Phalanger AST* analysis and overall *AST* design. First possible approach to overcome this is to introduce architectural changes which would make the compiler more extensible and loosely coupled, but from our experience with working with Phalanger codebase we have estimated the resource costs much above the research work itself. The second approach is to devise a complete specification of needed changes, and implement them with regards to Phalanger's design. This would be very hard without a proof-of-concept implementation.

Therefore, we have implemented a prototype implementation, which helped us to concentrate on control flow analysis and type analysis, instead of changing the complex implementation. In this chapter we will describe important features of implementation that were needed for implementation of mentioned analyses. Architecture and design is described in the next chapter since it contains many parts, which were not needed for implementation of analyses.

5.1 Language Features

Our implementation focuses on optimization of a program that is represented in *AST* form. Therefore we did not implement a parser, which would provide intuitive interface for the user of the prototype. The program for analysis is created manually in the source code by specifying its whole *AST*. We however did implement compiler back-end, i.e. emission of *CIL* object code for types and operations that were needed for demonstrating efficiency of the analysis. This helped us perform benchmarks which could be compared to results of other existing technologies, notably Phalanger and *HPHP*.

In order to simplify both control flow and type analysis, we have limited the *AST* to set of nodes similar to those presented in section 3.4 when we were describing local analysis algorithm. We chose to implement only the local analysis as global analysis as described in chapter 3 is similar and uses the same methodology that we will present.

AST nodes representing statements we have implemented are as follows:

- Script, which represents a single script file.
- Block statement, which represents sequence of statements.
- If statement, which represents basic branching statement.
- While statement, which represents natural loop.
- Break statement, which with If statement enables programmer to end the loop.
- Continue statement, which passes the control to the beginning of the loop.
- Assign statement, which assigns a value to a variable.
- Function call statement that calls a function and drops the return value.

Expressions we have implemented are as follows:

- Addition expressions that adds arbitrary number of expressions using PHP semantics.

- Concatenation expression, which concatenates arbitrary number of expressions, converting them to string beforehand.
- Opposite expression that returns an opposite of an expression.
- Multiplication expression, which returns a product of multiplication of arbitrary number of expressions.
- Reciprocal expression that returns multiplicative inverse of an expression.
- Remainder expression that returns a remainder of division of two expressions.
- Function call expression, that calls a function with given arguments.
- Logical And expression, which returns a conjunction of an arbitrary number of expressions.
- Logical Or expression that returns a disjunction of an arbitrary number of expressions.
- Logical Negation that returns a negation of a single expression.
- Equality expression that compares values of two expressions.
- Lesser Than expression, which compares values of two expressions.
- Constant expression, that evaluates to a given constant.
- Variable Use expression, that evaluates to a given variable.

Targets of variable access and function call are specified using binding nodes, which are as follows:

- Dynamic Global Variable Binding that binds a variable depending on its name.
- Dynamic Function Binding, which binds a function by its name.
- Static Global Variable binding, which binds a variable by given a static place in memory.
- Static Library Function that bind a function given its address in the memory.

Described set of features give us means for performing control-flow analysis and type analysis of the global code. Each scenario that we have tested had it starting with dynamic bindings and goal of analyzers and optimizers was to improve the overall performance.

Value types (not *CIL ValueTypes*) that we have implemented are:

- *Boolean*
- *Integer* (64-bit signed)
- *Float* (64bit floating point)
- *String* (CLR string)

Due to *PHP's* semantics, we use a union type Number, that represents both *Integer* and *Float* values. This is value type that is stored directly in memory. If a type could not be inferred (or type analysis was not done at all), we use boxed value for *Integer* and *Float* values. This is slower, but is actually used by *Phalanger* currently.

5.2 Analyses and Optimizations

We have implemented analyses and optimizations for limited model of *PHP* described above language. Since the model contains most of important features and *PHP* semantics, we can

5.2.1 Control Flow Analysis

Control flow analysis of the prototype uses almost the same algorithm and *Control Flow Graph* structure, which was presented in Chapter 3. Above described simplified model of *PHP* has the same

set of control statements. Because of this similarity we have chosen not to include detailed algorithm description herein.

A significant difference is explicit presence of virtual assign statements and virtual variables, which we have described previously, is not present as instead we use references of other nodes to expressions and therefore the original order of evaluation is kept. Type analysis stores its results globally and thus when it approaches a function call, it reuses results that it had already inferred for that node.

5.2.2 Type Analysis

For means of type analysis, we have defined type lattices as described in section 4.3.1 with the exception that we have simplified String lattice so is does not differentiate between arbitrary strings and strings convertible to numbers (hence having less complicated transition functions).

The analysis algorithm itself uses a control flow graph that was created by prior control flow analysis. It starts without any types assigned to variables, i.e. bottom value of the $Value$ lattice. It traverses edges of the flow graph in manner similar to *DFS* and upon reaching an assign statement it infers the type of assigned expression and finds its supremum with the last value inferred for the variable. This supremum is then used as new lattice type of the variable. *AST* is traversed until a fixed point of variable typing is reached.

Formal Description

For formal description, we need a set V of all CFG nodes, $E \subseteq \mathcal{P}(V \times V)$ set of all CFG edges and a map $Forward: V \to \mathcal{P}(E)$ that gives set of all edges coming out of the particular node. The algorithm will gradually define a map of variable types $Type: I \to Value$, where I is a set of variable identifiers and $Value$ is a lattice similar to lattice defined in section 4.3.1 and a map of edge versions $Version: E \to \mathbb{N}$ that assigns natural number to each edge. Additionally we will use inference algorithm $Infer: (I \to Value) \to P \to Value$, where P is a set of all expressions. This algorithm implements language semantics and returns a lattice value for given typing of variables and expression. Algorithm $Analyze: Entry \to (I \to Value)$ is then defined in the following way:

```
1:  Let Analyze(e: Entry ⊆ V) → (I → Value) =
2:      Type ← ∅
3:      Version ← ∅
4:      v ← 0
5:      Let Traverse(n: V) =
6:          If n is LinearNode then
7:              t ← Infer(Type, n.Expression)
8:              If Type(n) ≺ sup_Value{t; Type(n)} then
9:                  Type(n) ← sup_Value{t; Type(n)}
10:                 v ← v + 1
11:         For each e ∈ Forward(n) do
12:             If (Version(e) < v) then
13:                 Version(e) ← v
14:                 Traverse(e.Target)
15:     Traverse(e)
16:     Return Type
```

Pseudocode 13 - Type Analysis Algorithm

The algorithm traverses edges of the control flow graph, while remembering a "version" of devised typing map. This version allows algorithm to stop while it is capable of analyzing loops within the program. When the algorithm is deciding whether to run recursively on another node, it checks whether it visited the edge in current version of type map definition. If it passes along the edge, it updates the version info.

Finiteness

Finiteness of the presented algorithm can be easily proven by using the nature of edge traversal and Knaster-Tarski theorem [**43**]. First, we prove that for a given v, algorithm finishes in finite time. Then we prove that v is increased finite number of times. When both of these conditions hold, the algorithm is indeed finite.

Proof for finiteness for a given v is trivial and similar to finiteness proof for DFS. We never go forward through one edge twice, because we update its version number upon passing through it. Had version not changed, we can do this only once for each edge. Since set of all edges in the control flow graph is finite the traverse algorithm for a fixed v finishes in finite time.

The next step is to prove that v does not get increased too many times. We have a finite set of variables I and v is increased each time the $Type$ value for a specific variable is updated. Update of the $Type$ map is monotonous since we assign supremum of the new and the old value. Thus if we define a special vector lattice $Value^{|I|}$ that contains an inner lattice for each variable in I, we also get that $Traverse$ is monotonous (preserves order) in regards to lattice $Value^{|I|}$.

Then we can apply Knaster-Tarski theorem, which states that if we have a complete lattice L and order-preserving function $f: L \to L$, then the set of fixed point of f in L is also a complete lattice. Since complete lattices cannot be empty, this guarantees an existence of a fix point. Because $Value$ lattice has a finite path (using the lattice relation) from bottom to top, the $Value^{|I|}$ also have such

path finite. Therefore, there are a finite number of steps to achieve a fix point, existence of which is guaranteed. Hence, maximum value of v is finite.

Correctness

We cannot include detailed description of the correctness proof, since we have not formalized the $Infer$ function that encodes semantics of PHP language. However, given the nature of the algorithm described before, especially the fact that it achieves fixed point for order-preserving transition function, the correctness of the described algorithm itself is apparent when we presume correctness of the $Infer$ function.

The goal of the algorithm is to achieve such definition of the $Type$ map, which would cover all possible values that can occur in a particular variable at runtime. The $Infer$ function tells us, what values could be in the variable when other variables are typed using the current value of the $Type$. The resulting type is then compared with the current typing of the assigned variable and a supremum of both is assigned to the $Type$ map if they are not equal. When the algorithm ends, this system is stable, i.e. each additional use of the $Infer$ on any variable returns a lattice value lesser or equal to the value stored in the $Type$ map. This means that the algorithm does not break correctness of the $Infer$ function.

Since all values that are inferred are at least as high in the lattice hierarchy as $Infer$ function tells us, we can use the types that match inferred lattice values in runtime for optimizations. Again, we are presuming correctness of $Infer$ function and that it matches runtime behavior of values.

5.2.3 Optimizations

Optimizations in the implemented prototype are straightforward and use the $Type$ map inferred by type analysis algorithm. Because of the time constraints, we have managed to implement the optimizations only for Boolean, Integer, Float, Number and String types.

Constant Propagation

Constant propagation is trivial data-flow analysis that is used in most compilers [**31**], that replaces an expression with only constant operands by its value (in case all operators in it are referentially transparent). In our case, it is a by-product of the type analysis since we have included constant values in the $Value$ lattice and our implementation of $Infer$ tries to keep the values constant whenever possible.

The actual optimization is done by iteratively computing a lattice value after the $Type$ map was inferred in all leaf-most expressions that are not a call expression. We cannot compute a result of a call expression with constant arguments since we are not able to prove that a function is referentially transparent[12]. After a return value of an expression is guaranteed to be constant, we replace the expression with such constant.

Early Binding

Related to constant propagation is early binding algorithm, which enables the compiler to remove the overhead associated with late binding – especially the hash table access. We can remove the late bindings (Dynamic Global Variable Binding and Dynamic Function Binding in the prototype) if and only if their identifier expressions are constant values (which may have been result of the constant propagation).

[12] Proving that a function is referentially transparent is not difficult, but we chose not to include it in this thesis.

The dynamic binding is then replaced by a static binding that points to a specific internal the representation that would be used at runtime. In case of our CIL emitter, we use FieldInfo and MethodInfo objects, which represent fields and methods at runtime. The actual binding to memory location is done by JIT compiler.

Type Optimizations

As we have mentioned earlier, the dynamic representation in prototype is equivalent to the representation used by *Phalanger*, i.e. stored in a reference to an object. Value types are wrapped in a typed envelope called box. Boxing introduces performance penalty since each time a value is to be used, it's type need to be determined. This determination of type is inherently slower since it requires indirect access to the type description and then indirect reading of the value. The goal of type optimizations is to remove complexity of these tests.

Since in *PHP*, *Integer* values can overflow to *Float* values, we will be often getting lattice values that represent both the *Integer* and the *Float* – a *Number*. *Number* has two states indicated by a *Boolean IsDouble*. This means that we need to emit a branching instruction each time we are accessing it, but this is in general faster than fully dynamic values.

Our implementation provides specialized operators for *Integer*, *Double* and *Number* types. Later, we will show that such optimization helps a great deal in code that uses arithmetic operators.

5.3 Summary

In this chapter, we have briefly described implementation of algorithms and theory provided in the last two chapters. We have implemented basic type analysis for PHP-like language, in an easily extensible way. In the next chapter, we discuss architecture of the prototype in more depth including general design concepts we didn't implement. In chapter 7, we perform benchmarking of the prototype on several benchmarks.

We have implemented local optimization of PHP programs, concentrating on optimization that can be done due to inferred types. First class of optimizations relieve us of runtime type-checking in cases it is not needed. Second class of optimizations manages to perform operation with known operand types more efficiently.

6 AST-based Compiler Design

This chapter summarizes our results on AST-based compiler design we have devised while we were designing and implementing prototype. Since these results are not in line with direction of this thesis, we have included them in a separate chapter. The prototype implementation as found on the enclosed CD contains a partial implementation of ideas and algorithms mentioned herein.

Phalanger is implemented in a tightly coupled manner and therefore is not easily locally extensible as all extensions of functionality lead to a cascade of changes. During the design of the prototype we have emphasized the need to keep major architectural decisions in line with Phalanger's architecture in order to make future implementation of our work in Phalanger fluent. On the other hand, we have stated several requirements that we deemed useful if fulfilled in order to overcome design flaws of current implementation or keep the prototype easy to change. These are as follow:

1. **Loosely coupled design**. Phalanger, as open-source project, heavily suffers from the tightly coupled design because it is very hard to understand dependencies within the compiler and runtime, which makes very hard for new developers to make significant changes to the code without breaking entirely different part of the project. Additionally in order to make the project suitable for enterprise environment loosely coupled design is beneficial because of emerging possibility to make customer-specific improvements and changes.
2. **AST-centrism.** Because of CLR environment, low-level part of compiler backend diminished in managed compilers, or is not present at all. This makes AST-based approach to compiler design possible and in our opinion even more feasible because of simplicity of single-representation compilation, expressiveness and easier implementation of high-level analysis, which is more significant for PHP than for static languages. Phalanger has already taken this approach but it still uses a notion of global tables and contexts instead of using the AST to store the data in more localized manner, which more fits with the nature of dynamic language compilation.
3. **Parallelism.** Phalanger is currently designed as a single-threaded compiler and is not prepared for parallel compilation. Since static analyses of a dynamic language program often need to traverse many modules of the compiled program in order to infer an optimization, the notion partial compilation becomes less usable than in static languages and parallelism becomes beneficial. Moreover, the analyses presented in previous chapters are computationally very intensive.
4. **Modularity.** Most analyzes, which we have previously described exploit the fact that in an end-product application we know how the code is used. However, if are compiling a redistributable library we do not have this information and by direct PHP to CIL compilation we lose possibility to optimize the code in the final application. Better approach is to treat AST as an intermediate representation which can be serialized and redistributed by itself, enabling further optimization of target application.
5. **Laziness.** Static analyses and optimizations referenced and described in this thesis are usually dependent on results of other analyses. Usually, there are specific optimizations that use results of one or more static analyses, but often it is not possible to tell whether results of any analysis will be needed in the end. Therefore, the implementation would need means for analysis to be performed lazily on-demand, which would save time and memory.

Moreover this approach would make it easier to handle AST transformations that are often done by optimizations.

6. **Memory Conscience.** Current implementation of Phalanger does not handle memory requirements and keeps the whole AST in the memory. While this does not demonstrate itself in typical codebases, from our experience, some larger codebases are approaching the 32-bit memory limit (1.5 GB can be typically taken as a limit 32-bit .NET programs) and there are even larger codebases, that would surpass memory pool of currently typical client computers. Moreover, presented analyses can worsen this several times.

These requirements are not directly related to goals of this thesis. However, in our opinion they were worth of exploring along the way, since internal implementation and manageability are of concern in any software project. Extensible semantics are becoming a future goal for language compilers[13] and flexible analysis subsystem, which we are discussing in the rest of this chapter, will be of great use in such compilers.

We base our implementation on careful design of the basic operations or tree analyses and optimizations – analysis results and optimization transformation. Analysis results are solved by properties attached to the AST node through a fast interface would allow lazy invocation and automatic invalidation of results. Optimization transformations are done using shadow sub-trees that allow fast and parallel work with the AST.

6.1 AST Primitives

Our design contains four main primitive objects that together form the AST: nodes, links, connections and references. Basic relationships of these primitives can be seen in Figure 28. The three additional primitives help the compiler subsystem to perform common tasks that won't be otherwise possible or would have to be made independently by each algorithm. We will discuss these principles later.

Nodes represent the tree's actual nodes and contain links specific data. The AST has its root in compilation root node, which has top-level program primitives such as function definitions, type definitions or script files as its child nodes. Nodes have more abstract meaning than in traditional AST, for example the function call node has function binding as it's child node – which can be late bound (represented by callee's name) or early bound (represented by a reference to specific function definition).

Links are named groups of child nodes with the same meaning that are ordered. This generalizes single and multicast child references and provides the compiler subsystem a unified access to children of a specific node, without having an implementation of the visitor pattern support each node separately. Internally, links are indexed for easier manipulation, but an index of a child within a link is not meant to have semantic meaning.

[13] We know that C# 6.0 will have some notion of extensibility in terms of semantics and syntax. We cannot cite the source since this is internal information from Microsoft employees and was not yet revealed to the public.

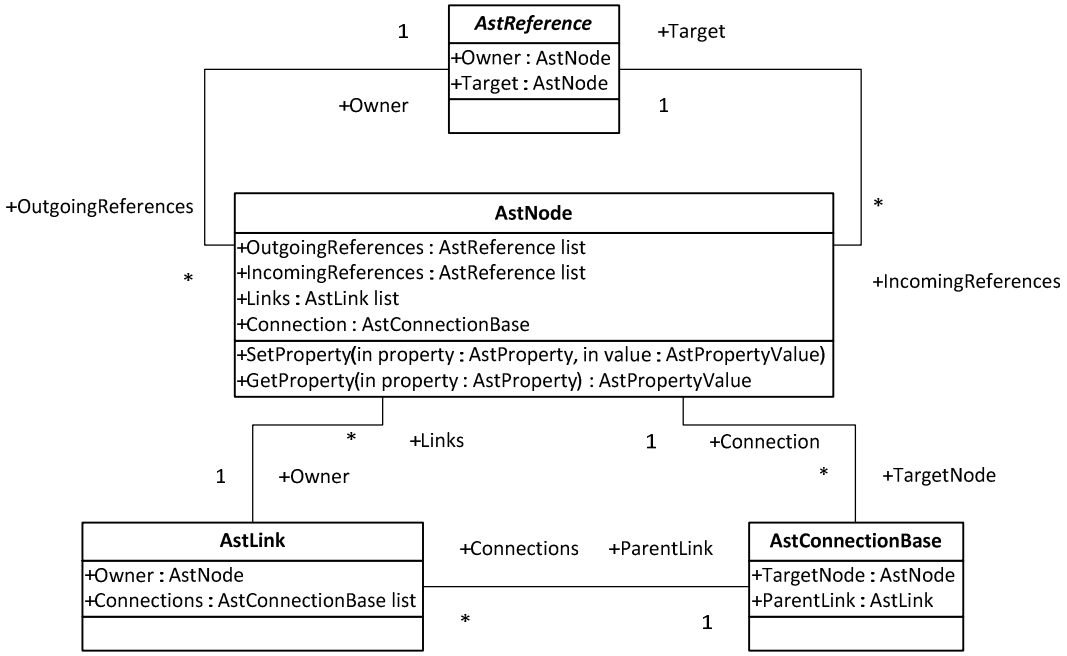


Figure 28 - AST primitives

Connections are an abstraction of relation between link and child node. There are four types of connections – normal connection is present everywhere in the AST and always has parent and target set. Root connection is present only in the compilation root node and does not have a parent set. Last two connection types are the reason for the abstraction – shadow parent connections and shadow child connections, which form boundaries of shadow trees, structures that allow efficient implementation of tree transformation algorithm, which will be discussed later.

Lastly, references are managed non-tree edges that point to hierarchically unrelated branches of AST, such as target of a jump statements or binding of a function call. The reasons for this abstraction are properties attached to AST nodes by tree transformation algorithms. These properties are usually dependent on other nodes and AST transformations can break validity of their values. Without managed references, non-tree edges in the AST will not be observable by the compilation subsystem and the results could not be automatically invalidated or corrected.

6.2 Analyzers and Optimizers

We have two main primate objects that deal with optimizations of the AST. Analyzers use portion of AST to infer a result that will be used later by other analyzers and optimizers. Optimizers transform a portion of AST into a different form, removing some nodes and adding new nodes. We have originally wanted to implement both principles within a single object, but we found it very difficult since optimizers operate in parallel while analyzer algorithms are usually hard to parallelize. Additionally, optimizers are usually dependent on analyzer results and this dependency would cause asynchronous optimizers to synchronously wait for analyzer results. This would lead to lower efficiency of synchronization algorithms presented afterwards.

Each optimizer operates on a partial sub-tree of the AST, which is defined as connected induced sub-graph of a tree. Root of this sub-tree is called determining node, which has importance for analyzer

scheduling, which is discussed later. For implemented analyses the depth of a partial sub-tree is very low – between 1 and 3. Result of the optimizer is transformation of AST, usually its reduction.

Analyzers operate on AST freely and infer additional information about the AST, which is stored in attached properties that are described further. Some analyzers do not work on AST, but rather a secondary representation – control flow graph, dependency graph, etc. These structures are synchronized similarly to AST but we will not discuss the synchronization herein.

6.3 Attached Properties

Static analyses infer information that is later used by optimizations and other compilation elements. In order to store analysis results in unified and extensible way, we have added notion of attached properties, which are similar in a way to attached properties in WPF[14]. Conceptually, each analyzer adds its results to AST nodes that were analyzed. These results can be dependent on each other and different analyzers and optimizers can query them.

One of the requirements stated was loosely coupled design. Attached properties in other systems, such as mentioned WPF, are declared within the containing type, which would be an example of tight coupling since AST nodes would be dependent on analyzers. In order to overcome this, we have moved the property declaration to the analyzers themselves, which is more difficult to implement, but does not have the flaws mentioned. Then, the whole AST structure is not dependent on any analyzer or optimization. Moreover, analyzers are dependent only on AST nodes and other analyzers, properties of which they actually use.

Other requirement fulfilled by attached properties is lazy analysis. When an analyzer or optimizer is starting (or rather before it starts), needed attached properties of other analyzers are queried. After these results are inferred, the analyzer is truly executed. For complex analyzers this causes chain reaction that first analyses the whole AST. Because of memory limitations of this approach, we have added support for heuristics that compute analyzer results before the dependent analyzer is executed. While these heuristics may be similar to phased compilation at the first glance, they are not equal since when AST changes through transformation, the heuristics have to be used again. Scheduling of analyzers will be discussed later in more detail.

6.3.1 AST Transformations

An analyzer can change AST through a transformation that takes limited subtree and conservatively transforms it into a different limited subtree that atomically replaces the former one. This operation is only bound to the particular limited subtree and does not inhibit a similar operation being done at the same time elsewhere in the AST. This effectively solves the parallelization problem, since analyzers can be scheduled to operate in parallel in parts of AST that will not be changed by other operating analyzers.

For formalization, we need several definitions. Let $T = (V, E)$ be a tree graph. We say that $S \subseteq T$ is a *limited subtree* if and only if S is a connected subgraph in T induced by $V(S)$. We say that a node $D(S) \in V$ is *determining node* of a limited subtree $S \subseteq T$ if it is a root of S and we denote parent node of $D(S)$ as $P(S)$. By $depth(S)$ we mean a *depth* of S i.e. a longest route from root of S to a leaf node.

[14] Windows Presentation Foundation is user interface library developed by Microsoft for .NET Framework.

We say that $\mathcal{B}(S)$ is a set of child subtree bindings of limited subtree S if it is a set of all tuples (n, C), where n is a leaf node of S and C is subtree induced by child node of n. In other words it is a set of connected components that will remain in subtree of $D(S)$ when S is removed from it, coupled with parent of root node of each of respective component.

By shadow tree S' of a limited subtree S we mean a tree for which we define $D(S')$ as determining node of S' and $\mathcal{B}(S')$ as set of child subtree bindings, for which:

$$\forall (n', C) \in \mathcal{B}(S') \exists (n, C) \in \mathcal{B}(S') : n \in V(S')$$

Shadow trees along with their child subtree bindings are input of generic transformation algorithm. Each analyzer is given a node N and selects limited subtree S on which it will operate and which satisfies $N = D(S)$. It creates a shadow tree S' of S and defines $\mathcal{B}(S')$ for it. At this point, S' is a replacement of S in the AST. The replacement algorithm commences as follows:

1) Set $children\big(P(S)\big) \coloneqq children\big(P(S)\big) \setminus D(S) \cup D(S')$. This replaces S by S' in the AST but the shared child subtrees cease to be part of the AST at this point.
2) For each $(n, C) \in \mathcal{B}(S')$ do:
    a. Set $children(n) \coloneqq children(n) \cup root(C)$.
    b. Set $parent\big(root(C)\big) \coloneqq n$.

Real implementation of shadow trees would not specify $\mathcal{B}$ sets, but rather would use previously mentioned concept of shadow connections, which is equivalent representation which is in line with AST design. Parent shadow connection is a shadow counterpart of connection between a specific link $L \in links\big(P(S)\big)$ and $D(S)$, which replaces the target $D(S)$ with $D(S')$. Child shadow connection for $(n, C) \in \mathcal{B}$ is a shadow counterpart of connection between specific link $L \in links\left(parent\big(root(C)\big)\right)$ and $root(C)$, which replaces L with specific $L' \in links(n)$. By specific link we mean a link specified by the connection, since links are only semantically grouped child nodes and above presented formalization does not cover either them or connections.

Concept of shadow trees has several interesting properties. Most notable is inherent scalable parallelization – since the analyzers and optimizers work only on a limited subtree of an AST while the part above and under the limited subtree is never accessed by it. Most static analyses and optimizations can be implemented in a way that single step uses a limited subtree with small constant depth. This means that number of concerned nodes is asymptotically constant or linear with complexity of expression, which is for real world programs asymptotically constant in relation to number of nodes in the AST. Therefore, if the performance is dependent only on scheduling algorithm that assigns analysis steps to different threads and the problem scales well with number of parallel threads.

Other benefit is, that since the output of the analyzer are well-defined sets of removed nodes, changed nodes, unchanged nodes and new nodes, a cleanup associated with attached properties and references is much easier to achieve.

6.4 AST Access Synchronization

Scheduling algorithm described later needs to synchronize access to AST nodes in order to work correctly in case two threads start to optimize colliding limited subtrees. There are two possible approaches to synchronization – top-down or bottom-up.

To formalize locking operations on AST nodes, let N be a set of all nodes. We define lock predicate in the following way:

$$lock_{value}(n) \in \{0,1\}, \forall n \in N$$

Value $lock_{value}(n) = 1$ denotes locked node and $lock_{value}(n) = 0$ unlocked node. Additionally, we need to define lock owner map, which tells us determining node of the limited subtree that is locked node part of:

$$lock_{owner}(n) \in N, \forall n \in N \cup \{NIL\}$$

Owner map is rather technical, but is present in implementation for debugging purposes. Following invariant condition binds these two maps together:

$$\forall n \in N: lock_{value}(n) = 0 \Leftrightarrow lock_{owner}(n) = NIL$$

We define operation $LOCK(n, o) \in \{0,1\}, n \in N, o \in N$ that locks single node as follows:

1) If $lock_{value}(n) = 1$ then return 0.
2) As an atomic operation, do the following:
    a. If $lock_{value}(n) = 0$ then set $lock_{value}(n) \coloneqq 1$.
    b. Else return 0.
3) Set $lock_{owner}\ (n) \coloneqq o$.
4) Return 1.

We define operation $UNLOCK(n, o), n \in N, o \in N$ that unlocks a single node and for which we assert that $lock_{owner}(n) = o$, as follows:

1) Set $lock_{value}(n) \coloneqq 0$.
2) Set $lock_{owner}(n) \coloneqq NIL$.

Obviously, both operations do not break the invariant condition.

6.4.1 Top-down locking

In top-down approach, the analyzer tries to lock the whole limited subtree starting in determining node. If any of the locks cannot be obtained, it means that some other analyzer already obtained them and then all already acquired locks need to be released.

Operation $LOCK_{td}(S) \in \{0,1\}, S \subseteq N$ is defined as follows:

1) Set $L \coloneqq \emptyset$.
2) For each $n \in S$ do:
    a. Set $b \coloneqq LOCK\big(n, D(S)\big)$.
    b. If $b = 1$ then set $L \coloneqq L \cup \{n\}$.
    c. If $b = 0$ then do:

i. For each $m \in L$ do $UNLOCK(m, D(S))$.
ii. Return 0.

3) Return 1.

Operation $UNLOCK_{td}(S), S \subseteq N$, where S is a limited subtree in N, has following step:

1) For each $n \in S$ do $UNLOCK(n, D(S))$.
2) Return

From the definition of $LOCK_{td}$ operation it is obvious that threads do not need to block during whole limited subtree locking operation, which relieves the implementation of the algorithm from thread waiting overhead. Our implementation uses Interlocked class, which is fastest synchronization primitive available on .NET Framework. $UNLOCK_{td}$ operation can be done asynchronously since only one thread can unlock the node. This means that number of needed synchronized operations is linear with number of nodes in the limited subtree. Memory usage of top-down locking is linear with number of nodes in the AST, in particular it is 4 or 8 bytes per node, depending on processor architecture, for representation of $lock_{value}$ predicate.

6.4.2 Bottom-up locking

Number of lock operations needed to lock the whole tree can be reduced by bottom-up approach, which exploits the fact that optimizers work on limited subtrees of small limited depth. The locking algorithm then stores the lock information only in determining node and in small number of its ancestor nodes.

First we need to know maximum depth of a limited subtrees needed by all analyzers, which we denote k_{max}. Then each node n contains a counter $C(n) \in \{0 \dots k_{max} + 1\}$, which represents number of levels locked by a limited subtree S with $D(S) = n$ or 0 if there is no such subtree. Addionally we define counters $C_1(n), \dots, C_{k_{max}}(n)$, where $C_i(n)$ denote number of determining nodes i layers beneath n that are currently being locked by some limited subtree. Formally these maps are defined using following invariants:

1) $\forall n \in \mathcal{N}: C(n) > 0 \Rightarrow \left(\forall m \in \bigcup_{i=1}^{C(n)-1} children^i(n) : C(m) = 0\right)$
2) $\forall n \in \mathcal{N} \forall i \in \{1 \dots C(n) - 1\}: C_i(n) = \left|\{m \in children^i(n): C(m) > 0\}\right|$

The first condition defines layer counter and the fact that when it is not zero, corresponding number of layers of child nodes must not be locked. The second invariant bind together the layer counter and child lock counters, which must be equal to number of nodes, locked on specific layer of child nodes.

When optimizers is given a determining node $n \in \mathcal{N}$ for which it needs to inspect limited subtree of depth $k \in \{1 \dots k_{max}\}$, it can do so if $test_{bu}(n, k) = 1$. We define predicate $test_{bu}(n, k) \in \{0,1\}, n \in \mathcal{N}, k \in \{1 \dots k_{max}\}$ by following conditions:

1) $C(n) = 0$
2) $\forall i \in \{1 \dots k\}: C_i(n) = 0$
3) $\forall i \in \{1 \dots k_{max}\}: C\left(parent^i(n)\right) \leq i$

From these conditions it can be seen that for AST's with larger branching coefficient, the bottom-up approach needs to inspect significantly less nodes than top-down approach. For instance an

optimization that locks limited subtree S with two layers of nodes needs to visit $D(S)$ and $parent(D(S))$. To check whether the lock is possible it needs to check $C(D(S))$, $C_1(D(S))$ and $C\left(parent(D(S))\right)$. One can see that both visited nodes and checked values are not dependent on number of children of $D(S)$ node, which is extremely strong property.

Obviously, to facilitate the locking the C and $C_i \forall i$ maps need to be changed accordingly. Since we are in parallel environment, the process of changing these maps need to commence atomically, so that definitions of C and $C_i \forall i$ are not violated. The operation $LOCK_{bu}(S) \in \{0,1\}, S \subseteq \mathcal{N}$, where S is a limited subtree in $\mathcal{N}$, is defined in the following way:

1) If $test_{bu}(D(S), depth(S)) = 0$ then return 0.
2) Set $L \coloneqq \emptyset$.
3) Set $b \coloneqq LOCK(D(S), D(S))$.
4) If $b = 0$ then return 0.
5) Else set $L \coloneqq L \cup \{D(S)\}$.
6) For each $n \in \{parent^i(D(S)) : i \in \{1 \dots k_{max}\}\}$ do:
    a. Set $b \coloneqq LOCK(n, D(S))$.
    b. If $b = 0$ then do
        i. For each $m \in L$ do $UNLOCK(m, D(S))$.
        ii. Return 0.
    c. Else set $L \coloneqq L \cup \{n\}$.
7) If $test_{bu}(D(S), depth(S)) = 0$ then do:
    a. For each $m \in L$ do $UNLOCK(m, D(S))$.
    b. Return 0.
8) Set $C(D(S)) \coloneqq depth(S)$.
9) Do $UNLOCK(D(S), D(S))$.
10) For each $n \in \{parent^i(D(S)) : i \in \{1 \dots k_{max}\}\}$ do:
    a. Set $C_i(n) \coloneqq C_i(n) + 1$.
    b. Do $UNLOCK(n, D(S))$.
11) Return 1.

For unlocking, we define the operation $UNLOCK_{bu}(S), S \subseteq \mathcal{N}$, where S is a limited subtree in $\mathcal{N}$, as follows:

1) Set $C(D(S)) \coloneqq 0$.
2) For each $n \in \{parent^i(D(S)) : i \in \{1 \dots k_{max}\}\}$ atomically set $C_i(n) \coloneqq C_i(n) - 1$.

At a first glance, the reader can see that both $LOCK_{bu}$ and $UNLOCK_{bu}$ do less locking operations than their top-down counterparts. The $LOCK$ operation is the most expensive operation in these algorithms in terms of resources. When the operation is successful is successful, $LOCK_{bu}$ need to invoke $LOCK$ $(k_{max} + 1)$-times, $LOCK_{td}$ needs to invoke it $|S|$-times. All presented analyses can be implemented with respect to $k_{max} = 2$, thus when determining node has two child nodes, $LOCK_{bu}$ and $LOCK_{td}$ have similar performance. For leaves, $LOCK_{td}$ is always faster, but in other cases, $LOCK_{bu}$ is faster. In extreme cases, for analyses with limited subtree depth of 2 with 4 child nodes per node, the number of locked nodes is 3 in case of $LOCK_{bu}$ and 21 in case of $LOCK_{td}$.

Note that bottom-up locking does other operations than $LOCK$ and $UNLOCK$ that are not present in top-down locking – setting and reading C and $C_i \forall i$ along with overhead associated with these operations in real-world environment. However, the complexity of these operations is again linear with k_{max} and thus constant for set k_{max}. Moreover, these operations have negligible resource cost compared to $LOCK$ operation.

From memory perspective, it is obvious that $LOCK_{bu}$ consumes more memory albeit still keeping constant complexity per node when we treat k_{max} as constant. Each node needs $k_{max} + 1$ counters, size of which is dependent on k_{max}. For sane values of k_{max}, the C counter will suffice with one byte. In case of $C_i \forall i$ the theoretical maximum value is dependent on the arity of nodes. This would mean for top bound of 256 children i bytes of memory for $C_i \forall i$.

However, C_i represents number of limited subtrees locked i layers beneath the parameter node. Since efficient implementation will lock only one subtree per thread, range of each C_i is between 0 and number of parallel threads and usually one thread per processor. Thus, for desktop computers one byte per C_i will suffice. In .NET Framework, this will be represented by an array of bytes, and each node will have a reference to this array, thus $4 + k_{max}$ or $8 + k_{max}$, depending on the architecture.

6.5 Optimizer Scheduling

Although we have solved synchronization of access to AST by locking nodes, practice tells us that it is best to avoid conflicts between threads completely – that is, to maximize percentage of successful lock operations and minimize the unsuccessful lock operations. Each unsuccessful lock consumes system resources without executing any optimization step. Had we allowed threads to wait, the overhead of such synchronization would me much higher than cost actual optimizing steps. In this section we are going to describe heuristic algorithm for scheduling analyzer steps, called scheduler in the implementation.

Main goal of the scheduler is to prepare lists of analyzer steps for worker threads with minimal need for using synchronization operations and minimizing conflicts. Regardless of locking algorithm it targets to divide the AST into parts, which will each be processed by one worker thread. When a part is finished, other part is given to analyzer, until any free part is left. Then, remaining parts are again divided into sub-parts and divided among the worker thread. This process continues until the analysis is finished, i.e. state when there is no analyzer that would change the *AST* when given any node.

In order to avoid execution of analyzers on nodes that are not processed by the analyzer, each analyzer presents a set of node types which it is able to analyze. This saves large number of lock operation that would be otherwise wasted.

Additionally, scheduler supports phasing and ordering of analyses, which can improve final performance of analysis greatly. Since optimizing analyzers change the AST, some analyzers can produce results that are later thrown away by other analyzer. Because of this, analyses can be grouped into phases and ordered within each phase.

6.6 Summary

In this chapter we have presented our approach to AST-based compiler design. While we have partially implemented principles and algorithms presented above, there much work needed to be done to efficiently implement such architecture. We plan to use part of presented design in Phalanger in the far future.

7 Evaluation of the Prototype

We have performed an evaluation of prototype implementation described in chapter 5 on several experimental benchmarks. Due to the fact that the prototype implementation does not have a parser attached, we have implemented only four benchmarks, which form a representative set that tests all important features of the language.

We have tested our prototype separately with three different settings to demonstrate efficiency of prototype's optimizations and on consequent differences we will demonstrate how effective can type analysis be even when employed only in local scope. We could not perform interesting benchmarks of function specializations, because it was out of scope of this thesis (see 4.4.1 for a brief description). Even then, the results we are presenting are very interesting and we have certainly managed to prove our concept.

7.1 Testing Methodology

The computer configuration on which we were testing has the same configuration that was described in section 2.1. We have evaluated technologies mentioned in that section, with an exception being *Roadsend*, which we did not include because of inconclusive results. Because we aimed to show efficiency that is induced by type analysis, we have included three different settings of our prototype, each with different level of optimizations. Therefore tested technologies were (see 2.1 for more detail):

- Phalanger 64-bit, running on Windows Server 2008 R2.
- PHP 5.3 32-bit, running on Windows Server 2008 R2.
- HPHP 64-bit, running on Ubuntu Linux 10.10 64-bit.
- Type Analysis Prototype, Full Optimizations (TAP-FO) 64-bit, running on Windows 7 x64.
- Type Analysis Prototype, No Analysis (TAP-NA) 64-bit, running on Windows 7 x64.
- Type Analysis Prototype, No Optimizations (TAP-NO) 64-bit, running on Windows 7 x64.

Since we did not have a parser for our prototype, we have prepared syntax trees for these benchmarks that are almost equivalent to the source code of benchmarks. For reference, the reader should check *Program.cs* file in *Prototype* directory on the enclosed CD-ROM, which contains code implementing all benchmarks presented herein. Settings called *TAP-FO* included both type analysis and variable binding optimizations, which resulted in variables being early bound and not stored in a hash table. Setting *TAP-NA* included only variable binding optimizations and *TAP-NO* included no optimizations at all. In neither setting we did not optimize the result on CIL-level and all specialized operators are simplest implementations for given operand types.

Each benchmark was run five times and the minimum result was taken as a final result. We did not target to make the measurement statistically sound, because exact results are not needed and tested technologies have differences so distinctive that it cannot be a product of measurement error. More notably, since each technology has different internal representations and algorithms, the results are inherently biased and not directly comparable.

Each benchmark first performs a warm-up test, which have removed a bias of JIT on CIL – Phalanger and all settings of our prototype. Then, it performs given number of iterations (we specify the

number in each test), which are then measured. We did not compute per-iteration performance as such results would not have any specific meaning. For time measurement, we use built-in PHP function microtime, which returns UNIX epoch time in floating-point seconds. In Snippet 16 we show a template that is used by all benchmarks. Each benchmark replaces <<TEST>> by the test function and <<ITERATIONS>> by number of benchmark iterations.

```
1:      <<TEST>>(100);
2:      $time = microtime(true);
3:      <<TEST>>(<<ITERATIONS>>);
4:      $time = microtime(true) - $time;
5:      echo $time;
```

Snippet 16 - Common template for benchmarks

We had put the benchmarks into a function on purpose. Phalanger and HPHP are optimized better for usage of local variables than global variables and therefore benefit from it. Our prototype uses global variables since it does not implement functions, but their performance is similar to that of local variables (or slightly worse).

7.2 Benchmarks

We have implemented four benchmarks, which test various situations that can occur in PHP programs. It is important to note, that in a typical practical program spends most of the time in library functions and only 20%-50% of the whole program time is spent in an interpreted PHP code. Naturally, faster technologies have this ratio even more diminished. Our benchmarks do not take this into account and are targeted to test the raw computational performance of the tested technology.

7.2.1 Empty Loop Benchmark

The first benchmark is the Empty loop benchmark, which is similar to the benchmark in section 2.1.1 with the difference being se of argument variable instead of class constant. This will not benefit Phalanger as it has usage of class constants optimized. The benchmark code can be seen in Snippet 17. We iterate the benchmark 100 million times.

```
1:      function emptyloop($n)
2:      {
3:              $x = 0;
4:              while(true) {
5:                      if ($n < $x)
6:                              break;
7:                      $x = $x + 1;
8:              }
9:      }
```

Snippet 17 - Empty Loop benchmark source code

We have used a while loop instead of a for loop because our prototype does not support for loop. The type analysis in our prototype is able to infer type for the variable "*$x*" and the "$n" variable, which is a global variable in our prototype. Since nor Phalanger or PHP perform type analysis, which difference does not affect the results. In case of HPHP, the argument type is inferred and hence does not produce a bias too.

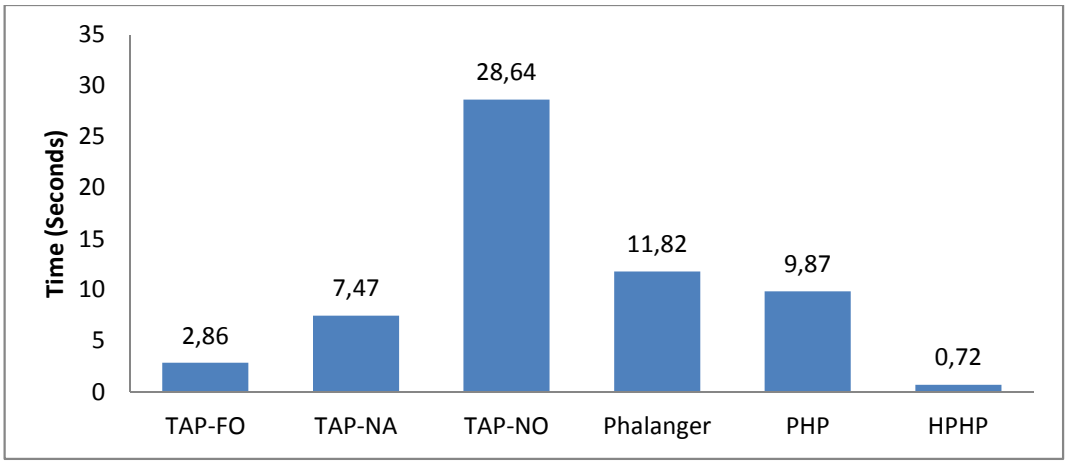


Figure 29 - Results of Empty Loop benchmark

Results, presented in Figure 29, show how inefficient is storage of variables in a hash map. If local variables are used, the performance improves almost four times. We attribute the fact that the prototype is faster than PHP and Phalanger even without type analysis to the precise modeling of PHP semantics, which is able to inherently optimize value types.

When the type analysis is enabled, it infers that "$x" has type *Number* and that "*$n*" has type *Integer*, which in turn speeds up all operators, variable reads and assign statements, because it removes boxing needed for generic values. HPHP shows best results, which we attribute to a pattern that removes *Number* semantics and leaves only *Integer*, which is much faster.

7.2.2 Euclid Algorithm Benchmark

Euclid algorithm benchmarks target to perform more arithmetical operations within a single iteration of the cycle. Since CPUs are able to optimize smaller blocks of code with less variables better than a more complex with many different variables, this approach is more near to real-world programs.

```
1:      function gcd($n)
2:      {
3:              $i = 0;
4:              while(true) {
5:                      if ($n < $i) break;
6:                      $x = 974788334;
7:                      $y = 160900432;
8:                      while(true) {
9:                              if ($y == 0)
10:                                     break;
11:                             else {
12:                                     $z = $y;
13:                                     $y = ($x % $y);
14:                                     $x = $z;
15:                             }
16:                     }
17:                     $i = $i + 1;
18:             }
19:     }
```

Snippet 18 - Euclid Algorithm benchmark source code

The benchmark is iterated 10 million times. We use predefined values for operands of the greatest common divisor search algorithm – this does not result in optimizations as no tested technology performs referential transparency analysis. All variables used in the inner cycle are whole numbers. The inner loop is passed several times before the result is achieved and the inner loop is left.

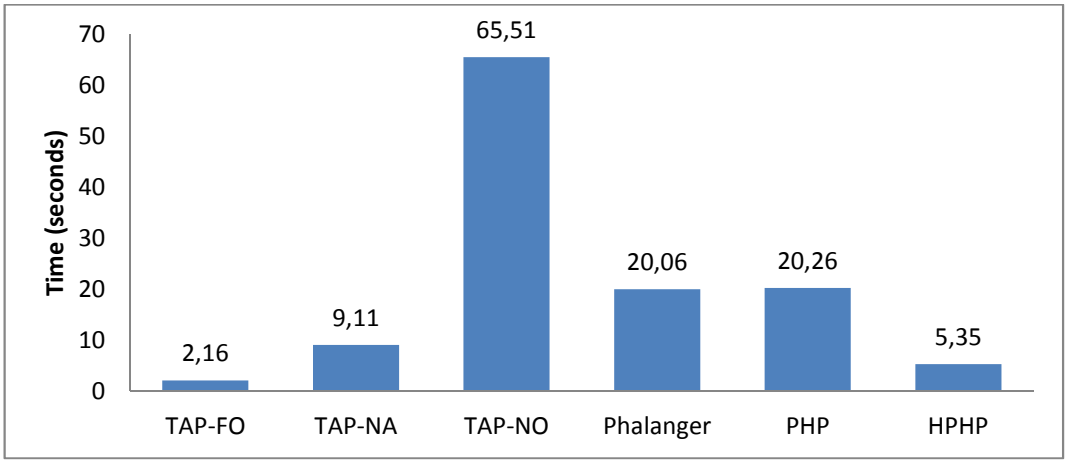


Figure 30 - Results of Euclid Algorithm benchmark

Results in Figure 30 again demonstrate the inefficiency of variables stored in a hash table. In these results, efficiency of our optimization-only version of the prototype is twice as fast as Phalanger. With full type analysis we are able to even surpass HPHP in performance, which we attribute to better a model of remainder operation.

7.2.3 Newton Approximation

The third benchmark is intended to test floating-point operation performance. We have implemented Newton approximation method of cubic polynomial function: $f(x) = x^3 - 2x^2 + \frac{1}{2}x - 5$. We are searching for a zero point and we iterate the algorithm until a necessary precision was achieved – $f(x) \in (-0.001; 0.001)$. We iterate the benchmark 1 million times.

```
1:      function num($n)
2:      {
3:              $i = 0;
4:              while(true) {
5:                      if ($n < $i) break;
6:                      $x = 544.443;
7:                      while(true) {
8:                              $fx = $x*$x*$x - 2.0*$x*$x + 0.5*$x - 5.0;
9:                              if ($fx > -0.001 && $fx < 0.001)
10:                                     break;
11:                             else {
12:                                     $dfx = 3.0*$x*$x - 4.0*$x + 0.5;
14:                                     $x = $x + -$fx / $dfx;
15:                             }
16:                     }
17:                     $i = $i + 1;
18:             }
19:     }
```

Snippet 19 - Newton Approximation benchmark source code

As obvious, this benchmark performs an extensive amount of computation per single iteration of the inner cycle. Our model of operations with *Float* operands result always in a *Float* value regardless of the argument type - the argument is converted to *Number*, *Float* or *Integer* and in all cases the value becomes a *Float* if the other operand is also a float. Because of this, we are able to infer that the most specialized valid type of "$x", "$fx" and "$dfx" variables is *Float*.

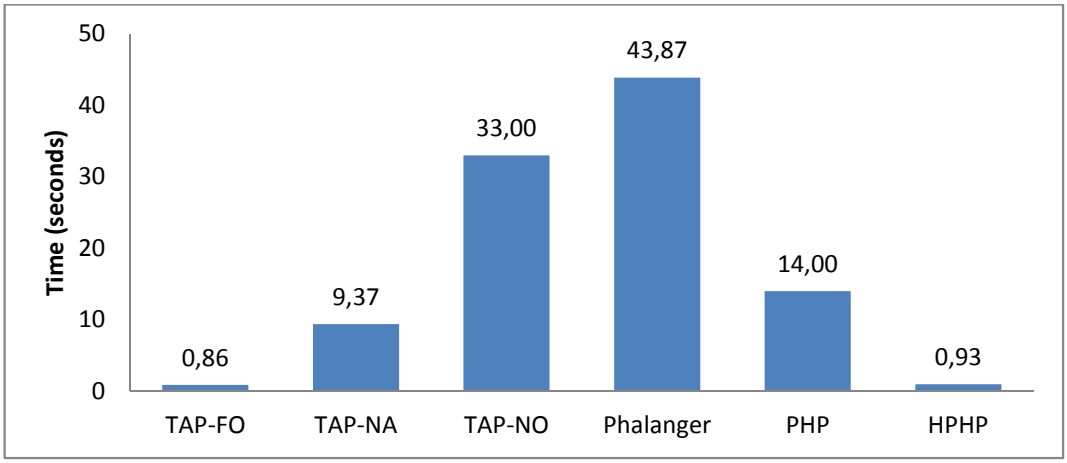


Figure 31 - Results of Newton Approximation benchmark

Results shown in Figure 31 indicate that Phalanger performs a large amount of unnecessary operations. This can be attributed to the complexity of expressions since *Phalanger* always boxes the result of single operation to unbox it few instructions after. Even without type analysis we were able to outperform *PHP* and with type analysis our prototype performs similarly to HPHP, which is probably able to perform almost the same amount of optimizations.

7.2.4 String Concatenation

The last benchmark we will present in this chapter aims to test the performance of string concatenation. This is a very frequent operation on web applications, which generate HTML in form of strings. Source code of this benchmark is included in Snippet 20.

```
1:      function foo($n) {
2:              $i = 0;
3:              while(true) {
4:                      if ($n < $i) break;
6:                      $s = "TESTBEGIN";
7:                      $j = 0;
8:                      while(true) {
9:                              if (20 < $j) break;
10:                             $s = $s."TESTPROGRESS"
11:                                     ."TESTPROGRESS"."TESTPROGRESS";
12:                             $j = $j + 1;
14:                     }
15:                     $i = $i + 1;
16:             }
17:     }
```

Snippet 20 - String Concatenation benchmark source code

The benchmark concatenates three string literals to a variable 20 times, resulting in 60 concatenations and a string of 729 characters in length. The prototype performs concatenations similarly to Phalanger using a *StringBuilder* object – a resizable array of characters in which the string are copied. PHP reallocates the memory in each step of concatenation. It is also important to note that both our prototype and Phalanger use UTF-16 string instead of binary string that are used by PHP and HPHP, therefore the two groups are not directly comparable as both operate on different amounts of data. We perform the benchmark 1 million times.

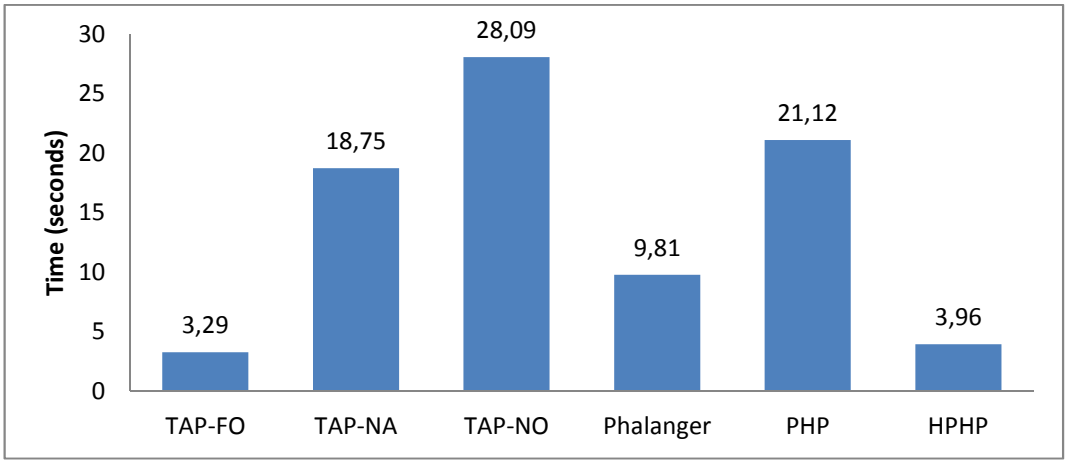


Figure 32 - Results of String Concatenation Benchmark

As obvious, Phalanger heavily optimizes string concatenation operator and manager to outperform original PHP and our prototype without type analysis two times. HPHP achieves even better results which we attribute to it joining the three constant strings that are concatenated with the variable. Our prototype does not simplify expressions in this way and even though we handle twice as much memory, the prototype has the best performance when type analysis is enabled. This is mainly because we are able to remove any virtual calls as we implement string operations for all combinations of operands.

7.3 Summary

Presented results show efficiency of our approach to type analysis of PHP programs. Different technologies are not directly comparable and we did not implement many possible low-level optimizations that can be done on CIL code that is output of our prototype. Even then, we were able to outperform HPHP, a project with more than 800 thousand lines of code and three years of development by one of the richest internet companies of today.

However, the difference between performance of implemented prototype with and without type analysis shows possibilities of the design. Especially, string concatenation improvement was completely unexpected as we have shown in section 2.1.9 that Phalanger's string concatenation is the fastest in typical scenario. Due to type analysis, we were able to remove a large amount of type-checking, resulting in almost static code for all presented benchmarks.

8 Future Work

We have achieved goals we have specified in the beginning of this thesis. Notwithstanding, the presented problem is very complex and we would like to include at least a short list of improvements and follow-up research that can be done about type analysis of *PHP* programs:

1) Extension of control flow analysis that would include exception handling and *goto* statements.
2) More detailed formalization of lattices especially of arrays. This would require recursive approach to express nested arrays.
3) Complete formalization of lattice transfer functions for type analysis.
4) Type analysis of object properties, which would help to improve performance of object-oriented applications.
5) Reverse data-flow in type analysis, i.e. an analysis of types to which variable values are converted. For example if we assign a string to a variable, which is subsequently used only as an integer, we can make the conversion beforehand.
6) Alias analysis that would be able to analyze situations when references are used. More important than optimization of access to references is ensuring that analysis behaves correctly.
7) Range analysis that would help the compiler remove *Number* semantics in many cases and would help to optimize *PHP* arrays that are used as regular arrays and therefore do not require a hash table.
8) Complete implementation in real-world compiler.
9) Test the concept on real-world applications.

Conclusion

In this thesis, we have focused on design and specification of static program analyses for Phalanger, a compiler of *PHP* language for *CLR*. We have identified features of PHP language for which their dynamic behavior causes non-negligible performance penalties. We have adapted existing control flow algorithm to conform to the nature of PHP and we have devised several techniques that perform type analysis of PHP programs. These techniques are capable of removing most of dynamic behavior and allow typical programs to have performance near that of static languages. While our results are targeted to PHP language, it is possible to adapt them for other dynamic languages.

We have proven our concept by implementing presented algorithms on a representative subset of the language, because implementation in the real-world compiler would be much more complicated and is of scope of this thesis. The simplified language contains most of important features that present problems for static program analysis and therefore results acquired in this environment can be generalized. We have shown that the performance of non-trivial programs is significantly improved compared to the existing Phalanger compiler.

The type analysis presented in this thesis has significant consequences. Instead of focusing on inferring a single type of a variable, which would be effective only for some local variables, we have focused on heuristic-based approach, which allow us to assign types even to properties of classes while keeping a possibility of using such class from outside with arbitrary values. Moreover, results of type analysis can be used for correctness analysis, informing the programmer about inconsistencies of inferred type information.

In the future, results of this thesis including the presented type analysis and results of the prototype implementation will be used for implementation of type analysis into Phalanger with support of full PHP semantics.

References

[1] PHP Group. PHP. [Online]. www.php.net

[2] Larry Wall. Perl Programming Language. [Online]. www.perl.org

[3] et al. Yukihiro Matsumoto. Ruby Programming Language. [Online]. http://rubylang.org

[4] Microsoft Corporation. IronPython - the Python programming language for the.NET Framework. [Online]. http://ironpython.net/

[5] Microsoft Corporation. IronRuby. [Online]. http://www.ironruby.net/

[6] Phalanger. [Online]. www.php-compiler.net

[7] Facebook Inc. HipHop for PHP. [Online]. https://github.com/facebook/hiphop-php/wiki/

[8] Paul Biggar, *Design and Implementation of an Ahead-of-Time Compiler for PHP*. Dublin, Ireland: Trinity College Dublin, 2009.

[9] Nikolas Askitis and Justin Zobel, "Cache-Conscious Collision Resolution," in *SPIRE 2005*, Buenos Aires, Argentina, 2005, pp. 91-102.

[10] Nikolas Askitis and Ranjan Sinha, "HAT-trie: A Cache-conscious Trie-based Data Structure for Strings," in *Conferences in Research and Practice in Information Technology (CRPIT), Vol. 62*, Ballarat, Australia, 2007, pp. 97-105.

[11] Jakub Míšek, Daniel Balaš, and Filip Zavoral, "Phalanger IntelliSense: Syntactic and Semantic Prediction," in *Informačné technológie – Aplikácia a Teória, ITAT 2009*.

[12] Adam Abonyi, Daniel Balaš, Miloslav Beňo, Jakub Míšek, and Filip Zavoral, "Phalanger Improvements," Department of Software Engineering, Charles University in Prague, Technical Report 2009.

[13] Jan Benda, Tomáš Matoušek, and Ladislav Prošek, "Phalanger: Compiling and running PHP applications on the Microsoft.NET platform," in *Proceedings of.NET Technologies 2006, the 4th International Conference on.NET Technologies*, Plzeň, Czech Republic, 2006, pp. 11-20.

[14] ECMA. (2006, June) Standard ECMA-335: Common Language Infrastructure. [Online]. http://www.ecma-international.org/publications/standards/Ecma-335.htm

[15] Edsko de Vries, John Gilbert, and Paul Biggar. PHC - The Open Source PHP Compiler. [Online]. http://www.phpcompiler.org

[16] John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *CACM*, April 1960.

[17] PHP Group. (2011, June) PHP History. [Online]. http://php.net/manual/en/history.php.php

[18] David Ungar and Randall Smith. Self Programming Language. [Online]. http://selflanguage.org/

[19] Microsoft Corporation..NET Framework Developer Center. [Online]. http://msdn.microsoft.com/en-us/netframework/default.aspx

[20] Xamarin. Mono Project. [Online]. http://www.mono-project.com

[21] Wayne Kelly. The Gardens Point Parser Generator. [Online]. http://plas.fit.qut.edu.au/gppg

[22] John Gough. The Gardens Point Scanner Generator. [Online]. http://plas.fit.qut.edu.au/gplex/

[23] Serge Lidin, *Inside Microsoft®.NET IL Assembler*.: Microsoft Press, 2002.

[24] Erik Meijer and John Gough. (2001) Technical Overview of the Common Language Runtime. [Online]. http://research.microsoft.com/~emeijer/Papers/CLR.pdf

[25] Miloslav Beňo, *Implementing the Dynamic Languages using DLR Technology*. Prague, 2010.

[26] Microsoft Corporation. Microsoft Dynamic Language Runtime. [Online]. http://www.codeplex.com/dlr

[27] Tomas Petricek. Using PHP objects from C# in a type-safe way. [Online]. http://tomasp.net/blog/ducktyping-in-phalaner.aspx

[28] Gavin Bierman, Erik Meijer, and Mads Torgersen, "Adding dynamic types to C#," in *June 2010 ECOOP 2010*, Maribor, Slovenia, 2010, pp. 76-100.

[29] Roadsend, Inc. Roadsend PHP. [Online]. http://www.roadsend.com/home/index.php

[30] Python Software Foundation. Python Programming Language. [Online]. http://www.python.org/

[31] Steven S. Muchnick, *Advanced Compiler Design & Implementation*. San Diego, California, USA: ACADEMIC PRESS, 1997.

[32] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The program dependence graph and its use in optimization," in *ACM Transactions on Programming Languages and Systems*, 1987.

[33] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently computing static single assignment form and the control dependence graph ," in *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 1991.

[34] INRIA. OCaml Programming Language. [Online]. http://caml.inria.fr/index.en.html

[35] Haskell B. Curry and Robert Feys, *Combinatory logic*. Amsterdam, Netherlands: North-Holland Pub. Co., 1958.

[36] Robert Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," in *Transactions of the American Mathematical Society*, 1969, pp. 29–60.

[37] Ole Agesen, "The Cartesian Product Algorithm," in *ECOOP'95 Conference Proceedings*, Århus, Denmark, 1995, pp. 2-26.

[38] Michael Furr, Jong-hoon An, Jeffrey S. Foster, and Michael Hicks, "Static Type Inference for Ruby," in *SAC'09*, Honolulu, Hawaii, USA, 2009, pp. 1859-1866.

[39] Mozilla Foundation. JavaScript. [Online]. http://developer.mozilla.org/en/JavaScript

[40] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. [Online]. http://cs.au.dk/~amoeller/papers/tajs/paper.pdf

[41] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich, "Effective representation of aliases and indirect memory operations in ssa form," in *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, Linköping, Sweden, 1996, pp. 253–267.

[42] Stanley N. Burris and H. P. Sankappanavar, *A Course in Universal Algebra*.: Springer-Verlag, 1981.

[43] Alfred Tarski, "A lattice-theoretical fixpoint theorem and its applications," in *Pacific Journal of Mathematics 5:2*, 1955, pp. 285-309.

[44] Saurabh Sinha and Mary Jean Harrold, "Control-Flow Analysis of Programs with Exception-Handling Constructs," Ohio State University, Columbus, OH, USA, Technical Report OSU-CISRC-7/98-TR25, 1998.

[45] Jens Palsberg and Michael I. Schwartzbach, "Object-Oriented Type Inference," in *OOPSLA'91*, Phoenix, Arizona, USA, 1991, pp. 146-161.

[46] Jakub Míšek and Filip Zavoral, "Syntactic and Semantic Prediction in Dynamic Languages," in *The proceedings of SERA 2009, Studies in Computational Intelligence, Springer Verlag*, Haikou, China, 2009.

[47] Moses Schönfinkel, "Uber die Bausteine der mathematischen Logik (Translated to English)," *Math. Ann.*, vol. 92, pp. 305-316.

List of Abbreviations

AST = abstract syntax tree.

BCL = Base Class Library.

CFA = control flow analysis.

CFG = control flow graph

CIL = Common Intermediate Language

CLI = Common Language Infrastructure

CLR = Common Language Runtime

DFA = data flow analysis

JIT = Just In Time

PHP = PHP Hypertext Preprocessor

TLS = thread-local storage

Appendix A: CD Content

- **Documents**
  - AdvancedOptimizationInDynamicLanguageCompiler.pdf
  - PhalangerUser.pdf
- **Binaries**
  - Phalanger 2.1 (March 2011) installer
    - Version we have benchmarked.
  - Phalanger 3.0 (December 2011) installer
    - Version with DLR-optimized dynamic operations.
- **Source Codes**
  - Phalanger 2.1 (March 2011) sources
    - Version we have benchmarked.
  - Phalanger 3.0 (December 2011) sources
    - Version with DLR-optimized dynamic operations.
  - TypeAnalysisPrototype sources
  - Benchmarks
    - Benchmarks that were used to evaluate the prototype.