Design and Implementation of an Ahead-of-Time Compiler for PHP

by

Paul Biggar

SUBMITTED VERSION
PLEASE DO NOT DISTRIBUTE

Dissertation
Presented in fulfillment
of the requirements
for the Degree of
Doctor of Philosophy

Trinity College Dublin

October, 2009

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Paul Biggar
18th October 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this dissertation upon request.

Paul Biggar
18th October 2009

Abstract

In recent years the importance of dynamic scripting languages — such as PHP, Python, Ruby and Javascript — has grown as they are used for an increasing amount of software development. Scripting languages provide high-level language features, a fast compile-modify-test environment for rapid prototyping, strong integration with database and web development systems, and extensive standard libraries. PHP powers many of the most popular web applications such as Facebook, Wikipedia and Yahoo. In general, there is a trend towards writing an increasing amount of an application in a scripting language rather than in a traditional programming language, not least to avoid the complexity of crossing between languages.

Despite their increasing popularity, most scripting language implementations remain interpreted. Typically, these implementations are slow, between one and two orders of magnitude slower than C. Improving the performance of scripting language implementations would be of significant benefit, however many of the features of scripting languages make them difficult to compile ahead of time. In this dissertation we argue that ahead-of-time compilation of scripting languages is both possible and valuable. We present `phc`, an ahead-of-time compiler for the PHP language. We describe the design and implementation of the compiler, and identify specific challenges in the design of a compiler for a dynamic scripting language.

We determine that there are three important features of scripting languages that are difficult to compile or reimplement. Since scripting languages are defined primarily through the semantics of their original implementations, they often change semantics between releases. They provide C APIs, used both for foreign-function interfaces and to write third-party extensions. These APIs typically have tight integration with the original implementation, and are used to provide large standard libraries, which are difficult to re-use, and costly to reimplement. Finally, they support run-time code generation. These features make the important goal of correctness difficult to achieve for compilers and reimplementations.

We present a technique to support these features in our ahead-of-time compiler for PHP. Our technique uses the original PHP implementation through the provided PHP C API, both in our compiler, and in our generated code. We support all of

these important scripting language features, particularly focusing on the correctness of compiled programs. Additionally, our approach allows us to automatically support limited *future* language changes. We present a discussion and performance evaluation of this technique, which we show increases the execution speed of PHP programs by 1.55x in our benchmarks. In order to improve this performance, static analysis and optimization are required.

However, static analysis of scripting languages such as PHP is difficult due to the features found in these languages. These features include dynamic typing with implicit type conversions, dynamic aliasing, implicit object and array creation, and overloading of simple operators. We find that as a result, simple analysis techniques such as SSA and def-use chains are not straightforward to use, and that a single unconstrained variable can ruin our analysis. We describe the challenging semantics of PHP, and a static analyser to model them. Our analysis combines alias analysis, type-inference and constant-propagation for PHP, computing results that are essential for other analyses and optimizations. Our empirical results show that our analysis is capable of determining that almost all program variables are unaliased, and that dynamic types for over 60% of variables can be statically determined by our analysis.

Acknowledgements

More than anyone, I'd like to thank David Gregg for his supervision, help and advice throughout the last six and a half years. He guided me through my undergraduate research, through my PhD applications, and throughout my PhD. I think it is fair to say that nearly everything I know about research comes from David.

Many supervisors are hands-off, David is certainly not. He has read everything I produced, he met me frequently to guide my research and his door is always open. His expertise on all manner of topics related to compilers and research has been invaluable.

It is the rare problem that David could not help me on. David made it his business to understand everything I did. I'm not sure he knew anything about scripting languages when I met him, but he spent a considerable amount of time and effort to understand my research thoroughly.

From the moment I started this PhD, David has been there for me. Any time there was a problem, he had time to help with it. In particular, at the start of 2007 I went through a particularly bad time. My then-current research wasn't going anywhere, and a change was needed. At the time, neither of us was really sure I'd make it to finish my PhD. During that time in particular, his help and guidance was very much appreciated.

Many of my colleagues merit thanks, in particular our research group: Kevin Williams, Nicholas Nash, Robert (Bobb) Crosbie, Jason McCandless and (Raymond) Scott Manley. They listened to my ideas and critiqued my work. Kevin also worked on scripting languages, and I learned a lot from chatting to him. Nick did an amazing job in our first paper, and I'm very proud of the result.

John Gilbert and Edsko de Vries were collaborators on phc. I sat in their office for the first year of my research, and I think I learned more about compilers from them than from any other source. Of course, they also started phc, and invited me to work on it, to which I owe them a great debt. I don't think the compiler I would have otherwise worked on would have been anything as successful.

I want to thank Edsko in particular. Working with Edsko is a lesson in how to code

well, and I learned a great deal from working on phc with him. He read a lot of my code, my commit logs, and all my papers. Arguing with Edsko helped me flesh out a great many ideas; no unclear idea gets past him. He also took on the herculean effort of reading my whole PhD in one night.

I also want to thank Kevin, Nick, Bobb, Jason, Scott, John and Edsko for their friendship. It would have been a long four years without it.

Jimmy Cleary interned with me during the summer of 2009. He worked on many aspects of phc, writing features, fixing bugs, and running a massive amount of experiments. His output far outstripped the time it took to train him in, and I honestly believe I would not have this PhD finished without his help.

Many people took the time to read my papers and dissertation, and offer comments, including David, Kevin, Nick, Bobb, Jason, Scott, Edsko, John and Jimmy. Thanks also to Nuno Lopes, Graham Kelly, Christopher Jones, Alex Gaynor, James Stanier, Cornelius Riemenschneider, and Etienne Kneuss, who all provided valuable feedback. Peter Hayes taught me the trick of reading my papers out loud. I believe this made a considerable difference to the quality of my writing.

I travelled considerably throughout my PhD. I learned a great deal from travelling to PLDI, ASPLOS and HOPL, from speaking to too many people to list here. Seminars and summer schools also taught me a great deal. Thanks to Markus Schordan, Sebastian Hack and Fabrice Rastello, and Koen de Bosschere for accepting me to Dagstuhl, the SSA seminar and the ACACES summer school, respectively. I learned a great deal, and met a lot of interesting people. Lectures from Mike Hind, Kathryn McKinley and Barbara Ryder were particularly interesting and inspiring.

I had many opportunities thanks to the kind help of others. I want to thank Dan Quinlan for his invitation to Livermore, to Ian Taylor for arranging the Google Tech Talk, to Brian Shire for having me to Facebook, and to Daniel Berlin for mentoring me on gcc during the Google Summer of Code. These seemingly small opportunities have benefited me much more than you might know.

My work was funded by the Irish Research Council for Science, Engineering and Technology funded by the National Development Plan. I would have been completely unable to work on a PhD without this funding. The Dun Laoghaire/Rathdown County Council paid my fees, and gave me a small stipend. I want to thank them for their help.

My family and friends have been very supportive throughout this PhD. No amount of thanks can express how much I value their friendship and love.

I could never have gotten to where I am without my fiancée, Orla McHenry. The last two years have been a lot of work and stress, and without her to welcome me home, I would have struggled to make it through. She brought me cookies when I worked late, cheered me up when I was down, and told me it would be fine when I worried. She was always there when I needed her. Knowing how proud she is of me always makes me smile. Of all of the things I've gained in the last four years, nothing makes me happier than Orla. I've never felt luckier than I am to have her.

Thank you all.

Contents

List of Figures

List of Tables

List of Listings

Introduction

1.1 My thesis

My thesis: Compiling scripting languages using an ahead-of-time compiler is possible and valuable.

The world of programming is currently undergoing a language renaissance. In particular, dynamic scripting languages are becoming incredibly popular, providing many features above those provided by large incumbents such as Java, C and C++. Perl is the language of system administration; PHP the server-side language of the internet; Javascript the client-side language the internet. Lua is currently the titleholder of embedded languages for scripting C and C++ applications, enjoying a particular niche in the gaming industry. Python and (to a lesser extent) Ruby are perhaps the future of scripting languages — powerful, elegant, generalist languages with staunch and devout followers, which are slowly replacing Perl and encroaching on PHP.

Despite this massive surge in interest, the research community has been somewhat silent on the issue. In personal conversations, I find many researchers do not see the point of scripting languages. As experts in many programming languages, nearly all of them statically typed, the dynamic typing which seems so freeing to the average programmer instead seems unappealing to the programming language researcher.

At the same time, the scripting language community cares little for 60 years of compiler research. Modern scripting languages are closer to two decades old than one, yet they are mostly implemented using interpreters (and often not-very-good interpreters). Although most enthusiasts yearn for speed in their applications, most of the scripting language communities solve speed issues be rewriting their performance sensitive code in C. Two steps forward, one step back.

My aim is to bridge the gap between these two worlds. This dissertation describes the design and implementation of `phc`, an ahead-of-time compiler for PHP. It aims to provide the basis for scripting language research, and show to the scripting language community that compilers are useful.

1.2 A Compiler for a Scripting Language

Most dynamic scripting languages are interpreted. This offers great flexibility and speed of development by removing the *compile* step. However, compilers are valuable tools for programmers.

The most important benefit which can be provided by a compiler is execution speed. Dynamic scripting languages have dynamic type systems, which require a lot of run-time type checks. PHP also has run-time reference behaviour, which must be checked on the majority of run-time value accesses. By providing static analyses at compile-time, a compiler can avoid many of these slowdowns.

Many important compiler optimizations make it easier for programmers to write programs. Optimizations such as constant propagation and loop-invariant code motion allow programmers to think about their programs at a slightly higher level, to avoid manual program optimization, and can reduce the cost of meta-programming facilities. Interpreted scripting language implementations force the user to be responsible for their own optimizations, which can decrease programmer productivity.

PHP is a language used mostly on the internet. One of the most important aspects of a web site's usability is its response time [Schurman and Brutlag, 2009]. However, a site's response time is affected by many systems: the server must be contacted over the internet, the web application run, the database consulted, the answer returned over the internet, and the web page must be rendered in the user's browser. This does not leave a large amount of time for the web application code to run.

Even in cases where PHP is not the bottleneck [Lerdorf, 2008], PHP programs are still limited in their ability to process data by the constraints of operating in a web environment. By providing a faster implementation, web applications are freed to do more in the same amount of time.

For other components of a web application, it is possible to improve performance. Databases access time may be decreased through faster hard drives, more memory and through caches such as `memcached`.[1] Network latency can be reduced by adding redundant servers and more bandwidth. Currently, the PHP execution time can only be decreased by increasing the speed of the processor, itself a challenge of late. Compilation provides another compelling means to do this.

[1] See www.danga.com/memcached/.

Large web companies use arrays of thousands of computers to host their websites. An increase in the speed of PHP of just 20-30% could free thousands of machines to be used for other purposes, to expand services to more users, or which could just be turned off. Compiler optimization can even save the environment.

Finally, compilers can be used in a vast range of tools to support programmers. Type-analysis, call-graphs and pointer information can aid code viewers, program understanding, integrated development environments (IDEs), program verification, and bug-finding tools. These have long been available for popular languages such as Java, but have been lacking for scripting languages because the frameworks for analysis have been unavailable.

1.3 Terminology

Terminology related to scripting languages can be confusing, as the *canonical* implementation of a language often has the same name as the language. The problems of canonical languages are discussed in great detail in Section 5. The Python community often refers to the canonical Python implementation as *CPython* to avoid this ambiguity.

In PHP, there are no existing conventions along these lines. In this dissertation I have used '*PHP*' to refer to the PHP language. The PHP language is defined by a canonical implementation which can be downloaded from php.net. This implementation I refer to as the '*PHP system*'. The `phc` compiler is another implementation of the PHP language.

1.4 Attribution

Most research is built upon the foundation laid by others, and this dissertation is no exception.

The `phc` front-end, which includes the scanner, parser, and the design of the AST, was written by de Vries and Gilbert [2007]. They are also the authors of `maketea` [de Vries and Gilbert, 2008], which generates code to support `phc`'s intermediate representations and provides the tools for their visitation and analysis. This is discussed at a high-level in Section 4.

De Vries was also a co-author of Biggar et al. [2009], a journal version of which is in preparation. This version is an earlier version of Chapter 5 in this dissertation. De Vries also wrote the original code generation in `phc`. The HIR and MIRs (see Section 4.2) were designed based partially on lowering passes which De Vries wrote.

Jimmy Cleary, a computer science undergraduate student at Trinity, worked as an intern in our research group during the summer of 2009. During this time, he:

- reengineered the HSSA form (see Section 6.5.5), fixing a large number of bugs, especially related to the dead-code elimination pass,
- fixed bugs in the *if-simplification* and *remove-loop-booleans* optimization passes, discussed in Section 4.6.2,
- wrote a pass to gather statistics from the optimizer, and scripts to convert those statistics into the graphs shown in Section 6.6.

The remaining work in this dissertation is mine. I:

- greatly expanded the lowering passes to handle a much larger portion of the PHP language,
- designed and implemented the HIR and MIR, described in Section 4.2,
- meticulously detailed the eccentricities of the PHP language, described in Sections 4, 5.2 and 6.2,
- came up with the idea to link the interpreter into `phc` and the generated code, described in Section 5,
- integrated the PHP system into the compiler, as detailed in Sections 4.5 and 5.4,
- rewrote the generated code to support more of the PHP language, and in a manner which could be optimized,
- wrote all of the optimizations described in Section 5.4.5,
- performed the experiments and evaluation in Sections 5.7 and 6.6,
- wrote the static analysis and optimization framework described in Chapter 6,
- performed all remaining research described in this dissertation, including surveying related work, and studying and describing the PHP language and PHP system. This forms a large portion of this dissertation.

1.5 Published Research

Parts of the research which appears in this dissertation have been published, are in submission, or in preparation in the following venues:

Published in peer reviewed publications:

- P. BIGGAR and D. GREGG, One Representation to Rule Them All, *PLDI '09 FIT Session*
- P. BIGGAR, E. DE VRIES and D. GREGG, A Practical Solution for Scripting Language Compilers, *SAC '09: ACM Symposium on Applied Computing (2009)*

In submission or preparation:

- P. BIGGAR, E. DE VRIES and D. GREGG, A Practical Solution for Scripting Language Compilers (Journal Version), *Submitted to Science of Computer Programming, July 2009*
- P. BIGGAR and D. GREGG, Static Analysis for Dynamic Scripting Languages, *In preparation for submission to ACM SIGPLAN 2010 conference on Programming language design and implementation*
- P. BIGGAR and D. GREGG, PHP Experience Report (working title), *In preparation as a chapter in Static Single Assignment Book (working title)*

Research talks:

- P. BIGGAR and D. GREGG, On the use of SSA with scripting languages, *Static Single-Assignment Form Seminar*, Autrans, France (April 2009).
- P. BIGGAR and D. GREGG, Compiling and Optimizing Scripting Languages,[2] *Google*, Mountain View, CA and *Lawrence Livermore National Labs*, Livermore, CA (March 2009).

[2] Available as a Google Tech Talk from youtube.com/watch?v=kKySEUrP7LA.

1.6 Structure of this Dissertation

This dissertation has the following structure:

- Chapter 2 introduces scripting languages, their characteristics, and existing research on their implementation.
- Chapter 3 introduces program analysis, and discusses existing research on alias analysis, type analysis, SSA, combined analyses, analysis of scripting languages.
- Chapter 4 describes the structure of `phc`, design decisions made during its development, and provides an experience report of creating a scripting language compiler.
- Chapter 5 deals with compiling *canonical languages*—languages defined by a single implementation. Scripting languages fall into this category. This chapter describes the first compelling means to combine ahead-of-time compilation with interpreted scripting languages.
- Chapter 6 deals with static analysis of PHP, particularly for optimization, and its implementation in `phc`. This builds on and corrects previous research on PHP static analysis, and is the most complete static analysis to date for the PHP language.
- Chapter 7 discusses my conclusions, potential future work, and remaining challenges in scripting language compilation.

CHAPTER 2

Scripting Languages

This dissertation discusses creating a compiler for PHP, a popular scripting language. A great deal of the information presented in this dissertation is applicable to other scripting languages, such as Perl, Python, Ruby, Javascript and Lua. This chapter provides background information on scripting languages, in particular discussing their features, type systems and programming styles. It also discusses the research into scripting language implementation, particularly run-time type-based optimizations for dynamic languages.

2.1 Background

This dissertation discusses analysis and implementation techniques for a set of *scripting languages*. In this section, I discuss the important features of languages in this subset.

The exact definition of a scripting language is somewhat unclear, as is the full set of languages which fall into this category. An early use of the term *scripting* [Ousterhout, 1997] names some scripting languages, and explains their use and advantages, but omits an attempt at a definition.

The simplest definition of a scripting language is a language designed or used principally to *script*, or drive the actions of a larger application. However, this definition is flawed, because popular scripting languages are used for application development, and many other languages have support for scripting.

The term '*scripting language*' is commonly used to describe languages with a common features set, discussed in this section. This is how the term is commonly used, and so we too use this '*definition*'. In this dissertation, the term *scripting language* refers to Javascript, Lua, Perl, PHP, Python, and Ruby, which are probably the most important languages for large-scale scripting today. Other common scripting languages which we do not discuss in this dissertation include Bash, Visual Basic, Tcl and ASP.

The term *scripting* implies a program which directs the actions of a larger program; a common term for a scripting language is a *glue* language. There are many ways

to script applications. For example, Javascript is most commonly used as a way to manipulate HTML pages in a browser. The HTML pages are exposed to a Javascript program via the Document Object Model [Keith, 2005]. A simple Javascript program might check that all the fields of a web form are filled in before submission to the server. More complex Javascript programs include email clients such as Gmail, or word processors and spreadsheet packages provided by Google Docs.

2.1.1 Uses of Scripting Languages

The scripting languages discussed in this dissertation have much in common. In particular, they are all designed and used for similar purposes, for scripting larger C/C++ applications, for shell scripting in the Unix environment, and for writing web applications. Table 2.1 shows the major uses of scripting languages, which languages support the uses, and which ones were designed for it.

Language	Embedding	Shell scripts	Web applications	
PHP	S	S	DP	Facebook, Yahoo, Wikipedia
Perl	SP	DP	SP	BBC, New York Times
Ruby	SP	S	SP	Twitter
Lua	DP	S	S	
Python	SP	SP	SP	Google, YouTube
Javascript			DP	All of the above

Table 2.1: *Popular scripting languages and how they are used;* 'Web applications' *indicates a language is used for scripting web application,* 'Embedding' *indicates the language can be embedded into C/C++ applications,* 'Shell scripts' *indicates it is used for shell scripting.* D *means the language is designed for the indicated purpose,* S *means the indicated purpose is supported,* P *means the language is popularly used for the indicated purpose.*

Scripting language implementations can often be embedded into C or C++ programs, to provide a scripting interface to an end-user. Lua is designed expressly for this purpose, and it is estimated that over 50% of video games shipped with a scripting environment use Lua.[1] Muhammad and Ierusalimschy [2007] provide a comparative analysis of these interfaces. Visual Basic is also used for scripting components of the Windows operating system.

Scripting languages are often used for *shell scripting*: to manipulate the output of Unix programs. Unix shells, such as `bash`, `ksh` and `zsh` are simple languages designed for this purpose. Perl was also designed for this purpose, in particular to replace the

[1] Mark de Loura surveyed approximately 100 '*game executives*', published online at gamasutra.com/blogs/MarkDeLoura/20090302/581/The_Engine_Survey_General_results.php.

lower-level tools `awk` and `sed`. It offers very powerful facilities for manipulating the output of external programs. Tcl also once filled this niche.

The largest and fastest growing area in which scripting languages dominate is in the design of web sites and web applications. PHP is currently available on more than 20 million web servers. It *scripts* the output of databases based on user input from HTTP forms. Ruby is also important in the area, using the Ruby on Rails framework. Python and Perl are also often used for programming web sites. Table 2.1 shows popular web applications built using each scripting language.

Finally, scripting languages are being used more and more for writing whole applications. Python in particular is being used for large applications, such as Mercurial, SCons, BitTorrent and Bazaar.

2.1.2 Scripting Language Feature Overview

Most scripting languages have a single canonical implementation, which is interpreted. The design of the languages and programs written in them often reflect this interpreted history; important language features include run-time code generation, a C API, variable-variables, dynamic class hierarchies, and other features which are straightforward to implement in an interpreter. Table 2.2 shows exactly which features are available in which scripting language.

Language	Different versions	`eval`	Variable-variables	C API	Duck typing	First-class functions	Dynamic class hierarchy
PHP	X	X	X	X	X	X	
Perl	X	X	X	X	X	X	X
Ruby	X	X	X	X	X	X	X
Lua	X	X		X	X	X	X
Python	X	X	X	X	X	X	X
Javascript	X	X	X		X	X	X

Table 2.2: *Popular scripting language features;* X *indicates the feature is present in a language.*

Scripting languages are often available in different versions. There are large differences between Perl 4 and Perl 5, and between PHP 4 and PHP 5. Scripting languages typically gain features over time, and often change semantics between major releases. Most scripting languages are around 15 years old, and there are many features that have changed between versions.

The scripting languages we discuss in this dissertation are all imperative, dynamically typed languages. No language we discuss requires or supports type declarations.[2] Only Perl even supports variable declarations.

Scripting languages all have facilities for manipulating the symbol-table, and for using strings as program code, at run-time. This provides support for accessing variables by name at run-time (*variable-variables*), as well as supporting the `eval` statement.

Scripting languages commonly include C APIs, which are part of the language. These C APIs are used as foreign function interfaces, and to write the languages' standard libraries. C APIs are discussed in more detail in Section 5, and compared by Muhammad and Ierusalimschy [2007].

An idiomatic scripting language feature is table support: all scripting languages have syntactic support for hashtables and use them as the principal data structuring unit. In fact, Python is the only scripting language to differentiate between lists and tables. The object models of Javascript, Lua, PHP, Python, Ruby are based on implementing objects using tables. As a result, most scripting languages are duck-typed (*duck-typing*), as discussed in Section 2.1.3.

From a language point of view, scripting languages store functions, method, modules, and classes in tables (*dynamic class hierarchy*). The tables allow these features to be replaced on the fly at run-time. Scripting languages also support first-class functions, but their use is not nearly as common as in higher-order or functional languages such as Scheme, Lisp, Haskell or OCaml. As such, scripting languages strongly resemble older dynamic languages: Ruby, Python and Javascript are very similar to Smalltalk and Self. However, PHP does not support dynamic class hierarchies, so we do not discuss them further.

Finally, most scripting languages have strong support for string operations, important for web application and shell scripting. Perl and Ruby both include syntactic support for regular expressions.

2.1.3 Scripting Language Type Systems

Many of the features which make scripting languages different from other programming languages relate to how they are *typed*. Informally, a language's type system is a

[2] Although PHP allows optional type annotations on function parameters, which are used for run-time checking.

means of specifying the values which are allowed in a language, and the operations allowed on those values. The facets of a scripting language's type system can be used to classify the language, and a scripting language's feature set are often affected by it's type system. Table 2.2 shows that the scripting languages we discuss are all dynamically typed, duck-typed languages.

Dynamic Typing

Informally, dynamic typing means that types are associated with run-time values. This is in contrast to static typing where types are associated with compile-time names such as variables or class fields. Commonly this means that a variable in a statically typed language may only hold values of a specified type, while a variable in a dynamically typed language may hold a value of any type. Dynamic typing is also known as *latent typing*.

The important difference between statically and dynamically typed languages is in type checking. Statically typed languages perform type checking statically, attempting to prevent errors before the program is run. Dynamically typed languages instead perform type checks at run-time. Arguably, as dynamic typing is used to mean dynamic checking, '*dynamic typing*' is a misnomer, but the term is in common use [Pierce, 2002].

Duck Typing

Dynamic typing in scripting language is also used to support *duck typing*. Duck typing is better described as a programming style than a type system, but it relies on dynamic language features commonly found in scripting languages' type systems.

Duck typing, as a term, comes from the Python community, although the concept is older. It is attributed to Alex Martelli, in a post to comp.lang.python:

> "In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with."

The argument is that although it is possible to dynamically check an object's class—in

the same manner as a nominal type system [Pierce, 2002, Section 19.3]—a better programming style is to instead rely on the object's run-time behaviour. The *type* of the object therefore depends on its *current* fields and methods, at the time that the object is used. As a result, the type system may support changing an object's fields and/or methods at run-time, including adding and removing them on a per-object basis.

Weak Typing

Weak typing is ostensibly the opposite of *strong typing*, though it has no formal definition. Both these terms are heavily overloaded, and it is not clear what they mean [Cardelli, 2004].

Weak typing is often considered to mean that values may be implicitly converted between scalar types, a definition commonly used in the scripting language community. However, other people use it to mean the same as dynamic typing, that unsafe conversions are allowed [Cardelli, 2004], that type conversions with an undefined meaning are possible, or that type checks are omitted. It is also used to mean that it is possible to evade the type system, for example, using C casts. Occasionally, it is used to imply that one language is inferior to another.

The popular meaning in the dynamic language community (implicit conversions) is perhaps a poor definition. Casts in most dynamic languages create new values from existing values, and are therefore safe. It is not possible to view the bits of a value of one type as another, as it is in C, where there are loopholes in the type system. Similarly, implicit conversions are well defined at run-time (though they are rarely specified in prose, a separate problem).[3]

Chris Smith's definitions[4] ring true with how I've most often seen the terms used:

> "**Strong typing:** A type system that I like and feel comfortable with."
>
> "**Weak typing:** A type system that worries me, or makes me feel uncomfortable."

Overall, because the term is not defined unambiguously, it is best not to use it. In this

[3] Thanks to Darren New for an enlightening discussion on this topic.
[4] http://www.pphsg.org/cdsmith/types.html

dissertation, I omit the term '*weak typing*', and instead refer to '*dynamic typing*' or '*implicit type conversions*' as necessary.

2.1.4 PHP

PHP was originally created as a domain specific language for templating HTML pages. PHP is a short form of its original name '*Personal Home Page/Forms Interpreter*'. It was since renamed to '*PHP Hypertext Preprocessor*'.

PHP has gone through several versions since its release in 1995. In this dissertation, we discuss PHP 5, the most recent version. The PHP language is described in detail in Section 6.2. Its canonical implementation is described in Sections 5.2 and 5.5.

The first versions of PHP were implemented in Perl. PHP's major influence is Perl, and it has similar syntax and semantics, including a dynamic type system with implicit type conversions, powerful support for hashtables, and garbage collection. PHP programs are typically web applications, and PHP is tightly integrated with the web environment, including extensive support for databases, web-servers, and HTTP and string operations.

PHP has a dynamic type system, with symbol-tables based on hashtables. PHP is not a fully duck-typed language. In particular, its classes are closed, and an object may not change its class once it is instantiated. While its fields are malleable, its class hierarchy is static, and an object's methods may not be changed at any time.

PHP is a hybrid object-orientation/procedural language, due to its history. Early versions of PHP did not support object-orientation, which was grafted into PHP 3. PHP 5 changed the object model so that objects were not copied by default. This legacy is still visible in the current language; in particular PHP references, discussed in detail in Section 6.2.11.

2.2 Run-time Type Feedback

A small amount of research has been performed on scripting language implementation. Most research on scripting language implementation has the aim of providing experience reports, instead of creating new implementation techniques. Section 5.3 describes a large number of scripting language implementations. This section focusses

instead on virtual machine research for dynamic languages, particularly in the area of feedback-based type analysis.

Run-time type feedback is used by just-in-time (JIT) compilers [Aycock, 2003] to optimize generated code based on type information collected at run-time. JIT compilers compile source or bytecode to native executable code at run-time. That is, their *run-time* and *compile-time* phases are intermingled. As a result, they can collect type information during execution, which can be used for optimization at compile-time. There are a number of run-time type feedback (hereafter *type feedback*) techniques available.

2.2.1 Research on the Self Programming Language

The Self research project [Ungar and Smith, 1987] produced three generations of run-time compilers, all of which used type-based optimizations. Many of Self's features are dynamic, and the Self JIT compilers were designed to execute these features quickly.

The first generation compiler [Chambers and Ungar, 1989] JIT compiled a method at a time. It optimized a significant amount of Self's dynamic features using compile-time type-inference. The type inference algorithm used was iterative type analysis [Palsberg and Schwartzbach, 1991], discussed in Section 3.4.2.

As a method's dynamic dispatch is expensive, it used *inline caching* to speed up the invocation. In this scheme, a call-site was backpatched with the address of the compiled method after compilation. The compiled method first performed type checks to ensure that it had the correct receiver—if not, the call was redirected towards the general dispatch mechanism. Agesen and Hölzle [1995] report this optimization to be successful 95% of the time. The first compiler could also inline small methods, and replace field accesses (which would typically involve a method invocation) with simple dereferences.

A number of other techniques were developed to support compilation based on types. If a variable could have multiple types after a control-flow join, *message splitting* is used to duplicate statements after the join. This allows variables to have single types in each statement, instead of merging the type information.

Type prediction compiles based on the idea that some methods are nearly always called with the same types. Arithmetic statements, for example, are called on integers 90%

of the time, according to benchmarks [Chambers and Ungar, 1989].

Finally, *maps* [Chambers et al., 1989] allow efficient implementation of duck-typing. As object fields are not declared, but added and removed at run-time, it is impossible to have a static object layout. To rectify this, objects are implemented using *maps*. Maps have a known number of fields, and the fields can be accessed with a single dereference. If a field is added to an object, its map (and therefore its type) changes. As a result, maps allow duck-typed languages to be compiled efficiently, in a similar fashion to class-based languages such as Smalltalk.

The second generation Self compiler [Chambers and Ungar, 1990] added a number of extra optimizations, including *extended splitting*. This was an extension of message splitting which allowed whole paths to be split. In particular, this had large advantages for the quality of type information available to the compiler in loops.

The third generation Self compiler [Hölzle and Ungar, 1994] used significantly more type feedback than before. Run-time types are recorded using a *Polymorphic Inline Cache* (PIC) [Hölzle et al., 1991]. This is an extension of an *inline cache* which can store more than one type.

Based on these run-time types, the compiler can optimistically inline a called method. Techniques from previous Self compilers are also improved with better type information. The compiler also recompiles code over time, based on improved type information, a technique also used in mixed-mode interpreters [Suganuma et al., 2002].

2.2.2 Comparison with Type Analysis

Type analysis, discussed in Section 3.4, is a complementary technique to run-time type feedback. Type analysis is typically performed ahead-of-time, and can lead to the removal of type checks and dynamic dispatch. In a perfect world, compile-time type analysis would lead to perfectly efficient code. In practice, because type analysis is a static technique, it must be conservative, and often cannot compete with the accuracy of run-time type feedback. However, run-time type feedback has a run-time overhead, when compared to type analysis.

Agesen and Hölzle [1995] implemented both type analysis and run-time type feedback. Their experiments report that both techniques lead to the inlining of over 95% of

method dispatches. However, type-feedback was determined to be significantly more powerful for integer programs. In Self, arithmetic operations between two integers can lead to overflow, in which case the type of the result of the operation would be a `BigInteger`. As such, pure type analysis cannot infer that the result of the operation is an integer. However, run-time type feedback can discern this. As a result, in Agesen and Hölzle's experiments, programs compiled with type-inference ran 70% slower than those compiled using type feedback.

2.2.3 Application to Scripting Languages

Compilation techniques designed for Self were mostly aimed at eliminating dynamic dispatch and efficiently compiling objects in a language where all types are objects. These optimizations are also important for dynamic scripting languages. Recent dynamic language implementations have reused the optimizations. V8[5] uses *hidden classes*, a run-time analysis equivalent to Self's maps, to efficiently JIT compile Javascript programs. Polymorphic Inline Caches are used by SquirrelFish Extreme[6] for the same purpose.

Scripting languages also use type analysis to generate efficient code for numeric computation. *Specialization* is a technique for efficient compilation of numeric values, and *traces* extend the granularity over which specialization can be performed.

Specialization

Psyco [Rigo, 2004] is a JIT compiler for Python which uses *specialization*. *Specialization* means compiling methods specially based on the types of their arguments. For example, a function called with two integer arguments would be compiled differently to the same function called with two floating point arguments.

The Psyco specializer differs from previous work, in that it does not use profiling or statistics. Instead, it runs completely lazily, compiling a function any time it is called with a new set of types. In addition, it also compiles a function lazily, stopping halfway, waiting until more run-time information information about a value is available by executing the function, and then continuing compilation. It then specializes based on this progressively more specific information about run-time types.

[5] V8's hidden class implementation is described at code.google.com/apis/v8/design.html.
[6] See http://webkit.org/blog/214/introducing-squirrelfish-extreme/.

Traces

A *trace-based* JIT compiler profiles a program, gathering type information at run-time. It uses this type information to optimize the generated code according to the actual types of the program. As such, it is very similar to the type-based feedback found in Self compilers, as well as Psyco's specialization. The important distinction of a trace-based JIT compiler is that it does not necessarily use functions or methods as its compilation unit.

TraceMonkey [Gal et al., 2009] is a trace-based compiler for Javascript, which is part of Mozilla Firefox. It records frequently executed (*hot*) sequences of instructions and compiles them to native code. This sequence of instructions—the *trace*—forms a loop, which may cross method boundaries. A trace has a single type associated with each variable in the program. The compiler speculates that this set of types will be static for future iterations, but it cannot be certain. As such, it must add guards at various points to exit the trace. These *side-exits* may themselves become hot, and be combined into an existing trace to form a *trace-tree*. A single tree may have many branches, depending on how mutable variables types are in the program. TraceMonkey reports speedups of 2–20x over the existing bytecode interpreter.

Type-based Interpreters

Although these techniques are typically applied to JIT compilers, type specialization can also be performed in interpreters. *Dynamic interpretation* [Williams, 2009] is a technique to record type information in a *Dynamic Intermediate Form* (DIR). It was implemented for the Lua scripting language.

The DIR is a combined control-flow and *type-flow* graph. Its nodes are bytecode instructions, and its edges are either control-flow or type-flow edges. Type-flow edges allow the interpreter to build complete program paths on which all the types of all variables are known. Each of these paths is built at run-time based on the concrete types in the program.

A DIR node is built based on the complete set of types before the statement. For each statement, it is known if they may cause type-transitions for any variable. For each type transition which occurs in practice, a node has a successor. Some nodes, such as arithmetic instructions, have only one successor. Hashtable accesses may return any type, so they have a separate successor for each of the nine basic Lua types.

After a statement is executed, the type of its result is used to determine its successor. If its successor node is not known, then a node with the correct instruction and set of types is sought. If such as node is not found, a new node is created. Each node stores the full set of types at that program point, so that it may be searched for, and so that it will know the types to search for in its successor. As long as there are no type changes in the program, no new nodes will be built.

Based on this type feedback, the dynamic interpreter uses type-specialized bytecode. The types of an operand are known when the node is built. The next time the node is executed, the operands are guaranteed to have the same types, and so type-checks are avoided. This technique is particularly effective on arithmetic statements, which have traditionally been slow to interpret. In his experiments, Williams reports a speedup of 2.6x in the best case. In the average case, their speedup is 1.45x. They achieve particular speedups on benchmarks implementing numeric algorithms and loops, but not on those that use method dispatch.

2.3 Summary

This chapter provides background information on scripting languages. It describes their feature set, how they are used, and their type systems. It discusses important literature on the implementation of dynamic languages, particularly related to run-time, type-based feedback. It compares it to type-analysis, and discusses how it is used in modern scripting languages.

The major focus of this dissertation is combining existing scripting language implementations with ahead-of-time static analyses. The next chapter discusses the latter in more detail.

Program Analysis

Program analysis is a technique for determining properties of a program. In this chapter, we discuss *static* program analyses. A static program analysis attempts to determine program properties ahead-of-time. In Chapter 2 we discuss some techniques which use run-time, or dynamic, analysis.

This chapter begins by introducing dataflow analysis. Alias analysis is discussed in great detail, and type analysis, combined analysis, and the SSA form are all discussed, These analyses form the basis for the static analyses we develop in Chapter 6.

3.1 Dataflow Analysis

The static analyses in this chapter can typically be considered to be *dataflow* analyses. A dataflow analysis can be generalized as one which statically models the flow of values in a program at run-time.

A dataflow analysis is formulated over a *control-flow graph*. A control flow graph (CFG) is built for each function in a program, and is a graph in which the nodes are *basic blocks*, and the edges represent the function's control-flow. A *basic block* is a maximal list of consecutive statements with one entry point and one exit point. The CFG has *entry* and *exit* blocks for the entry and return points of the function.

Analyses are often *set-based*, that is, they propagate sets of information. A dataflow analysis is classified by how the sets are propagated. An analysis is either *forwards* or *backwards*. A forwards analysis propagates program information from a block to its successor. A backwards analysis propagates program information from a block to its predecessor. From here on, we shall discuss forwards analyses.

At each block, an analysis combines its predecessors' sets, adds and removes elements from the set, and passes the sets to its successors. The predecessors sets are merged by either set union or set intersection. For each block, *gen* and *kill* sets are created, which add and remove information from the set. This process is typically organized using *transfer functions*, which specify which additional items should be killed and which

additional items should be added to the sets.

The *gen* and *kill* sets are used to statically approximate the run-time behaviour of a statement. The components of a statement which are mapped into those sets are called *lvalues* and *rvalues*. An *lvalue* is value which may be written to during a statement, and an *rvalue* is a value which may be read from during a statement.

As a concrete example, consider a *reaching definition* analysis [Aho et al., 2007, Section 9.2.4]. Reaching definitions at a point in the program are definitions whose values reach that program point. It is a forwards analysis, propagating sets of definitions from blocks to their successors.

To begin the analysis, all blocks are initialized to the empty set. Blocks have both input (IN) and output (OUT) sets. A block's IN set is the set union of all of its predecessors' OUT sets. A block's OUT set is determined with the following transfer function:

$$OUT = GEN \cup (IN \setminus KILL)$$

GEN are the solutions generated in a block. In this example, they are the set of definitions created in the block. The KILL set contains the definitions invalidated by this block.

The transfer functions are iteratively applied to all blocks. When no solutions have changed, the algorithm is said to have reached a *fix-point*, that is, no solutions changed. This is often implemented using a *worklist* algorithm, in which the successors of changed blocks are added to a list to be processed again. Most static analyses can be expressed using variations on this framework, even if they do not use sets.

In the next section, we look at alias analysis, an application of dataflow analysis which is central to this dissertation.

3.2 Alias Analysis

Alias analysis is a program analysis aimed at discovered program properties related to *aliasing*, defined by Landi and Ryder [1991]:

> "Aliasing occurs at some program point during program execution when two or more names exist for the same location."

A great deal of research over the last 20 years has been dedicated to the research of program analyses to detect aliasing. Alias analysis is related to *points-to analysis* [Emami et al., 1994], *pointer analysis* [Hind and Pioli, 2000], *mod-ref*[1] or *side effect analysis* [Landi et al., 1993] and *may-alias* information [Cytron and Gershbein, 1993]. In this dissertation, we shall refer to all of these as *alias analysis*, as the overlap between them is so great that it is not worth distinguishing between them for our purpose. Alias analysis also has close ties to *escape analysis* (see Section 3.2.3) and *shape analysis* (see Section 3.2.5).

Powerful alias analysis techniques became important due to the interest in analysing C and C++ programs. As a result, the study of alias analysis can be roughly split into four eras:

1. Prehistory
2. Ascendency: The formulation of important techniques, during the 1990s.
3. Diversion: A short time with a lot of research into escape analysis
4. Scalability: Recent research has focussed on scaling alias analysis to millions of lines of code

3.2.1 Pre-history

Pre-history refers to early research which shaped the development of alias analysis. A number of the important techniques described in Section 3.2.2 were developed during this time.

Work on alias analysis evolved from earlier work on Fortran. Fortran's memory model is significantly less powerful than that of C and C++, and it restricts the ability of the programmer to create aliases. As a result, Fortran did not require powerful alias analysis techniques. As research into C programs became more important, existing techniques of analysis were used as building blocks.

A significant aim of this work was to analyse heap memory allocated in a program, in order to apply scalar optimizations and allow reuse of allocated memory [Chase et al., 1990, Jones and Muchnick, 1982, Larus and Hilfinger, 1988]. Typically, these

[1] Modification/Reference analysis.

analyses operated on a graph of dynamically allocated objects, building the graph as the program was analysed.

The major contribution of this era was creating the building blocks that were used later on. The major focus of the research from this era was developing techniques for naming different objects created at the same allocation site, which is similar to the goals of modern alias analyses. Among a number of important developments was the ability to identify recursive structures such as linked-lists [Jones and Muchnick, 1982]. Larus and Hilfinger [1988] showed how to model strong updates when an analysis could prove that a variable, which is the basis for modern *flow-sensitive* algorithms. Interprocedural analysis [Chase et al., 1990, Cooper and Kennedy, 1988, 1989] and *k-limited* [Horwitz et al., 1989, Jones and Muchnick, 1979, Larus, 1989, Ruggieri and Murtagh, 1988] naming schemes, were both investigated, which are closely related to *context-sensitivity* and *object-sensitivity*, discussed in the next section.

This early work also provides a foundation for *shape analysis*, which attempts to identify recursive data structures such as trees and linked-lists [Jones and Muchnick, 1982]. Escape analysis—statically identifying the scope of heap-allocated memory—descends from work to detect the lifetime of objects in a program [Ruggieri and Murtagh, 1988].

3.2.2 Ascendency

The second era of alias analysis research began with attempts to model the aliasing behaviour of C. It can be considered to start approximately with Landi and Ryder's [1991]' classification research, in particular noting the complexity of analysing C as compared to Fortran. The era characterized largely by the systematic exploration of the parameters of powerful alias analyses.

A number of important alias analysis algorithms were developed in this time, with similar approaches. For example, we describe Andersen's [1994] analysis:

- Considering all statements in the program, identify their effect on the aliases between memory locations in the program. Each memory location is represented with a name, perhaps its variable name concatenated with function in which it is declared. Typically, a statement will add an alias between two names, or do nothing.

- Given all the aliases, solve the solution, creating a single set of aliases for the whole program. The result is a graph with edges between names which are aliases.

Burke et al.'s [1994] and Choi et al.'s [1993] algorithms are superficially different, using dataflow-style algorithms to iterate through the statements of the program, updating a solution set of aliases as they proceed. However, their net result is very similar: one or more graphs with names for nodes, and edges indicating possible aliases between the names, at particular points in the program.

The major analyses of the era differ in a number of important parameters, which affect their performance (this is, the speed of the analysis) and precision:[2]

naming-schemes: Many alias analysis algorithms differ in how they name memory locations. Different naming techniques strongly affect the speed and precision of an analysis. If a single name can refer to more than one piece of memory, that memory is *abstract*; otherwise it is *concrete*.

Two important components which affect how naming schemes are used are *context-sensitivity* and *object-sensitivity*.

context-sensitivity: Interprocedural analysis algorithms typically use either a *call-string* or a *functional* approach [Sharir and Pnueli, 1981]. A functional approach attempts to create a model of a function which takes inputs and generates outputs. A call-string approach names objects based in some way upon the callgraph at the point of allocation.

For example, if memory is allocated in function `foo`, on line 12, a context-insensitive approach might name the memory '*foo:12*'. If `foo` was called from two different functions, two pieces of memory would have the same name. In a context-sensitive approach, the two callers `bar1` and `bar2` would lend their names to the memory. The result would be two pieces of memory which do not alias, namely '*bar1:foo:12*' and '*bar2:foo:12*'.

Context-sensitive algorithms use calling context in their analyses, typically using the call-string approach. This allows it considers two pieces of memory from the same allocation site to be different for two different callers. Fully context-sensitive algorithms can take exponential time [Landi and

[2] Different analyses use different names for the same ideas; we try to use accepted terminology from existing survey papers [Hind and Pioli, 2000, Ryder, 2003].

Ryder, 1992], but context-insensitive algorithms can be much less precise, especially in the presence of factory methods (including C '*constructors*'). In general, the solution is to use a compromise, using only a small part of the call-stack in the name.

object-sensitivity: One important algorithm [Emami et al., 1994] chooses a single name to represent the entire heap, analysing only a function's local variables. Others use the object's static class—or concrete class if known—combining every object of the same type. Compared to the common technique of choosing a separate name for each allocation site, these techniques offers some performance benefits. With only one name per class, the total number of names may be significantly lower than using one name per allocation site.

flow-sensitivity: A flow-sensitive algorithm considers program flow, with new definitions overriding old—these new definitions are *strong-updates* or *killing-definitions*. Flow-sensitive algorithms must iterate until reaching a fix-point, reducing performance, especially when combined with context-sensitivity. A flow-insensitive algorithm removes any ordering between statements in a function, and can only use *weak-updates*, sacrificing precision for performance.

field-sensitivity: Objects with fields can be analysed by representing each field individually for precision, or by limiting the number of fields represented (possibly to just one) for speed.

directionality: Given an assignment statement, an analysis using *unification* [Steensgaard, 1996] will consider the lvalue to alias the rvalue, thereby sharing alias sets. An analysis which uses *inclusion* [Andersen, 1994] will only consider that the rvalue's alias set must be a subset of the lvalue's set. Therefore inclusion is more precise, but unification has greater performance, due to its symmetry.

representation: The size of the result set of an alias analysis can depend on its representation. Alias pairs [Landi and Ryder, 1991] consist of tuples with a list of names which can alias. They represent the tuple $\langle p,q \rangle$ separately from $\langle *p,*q \rangle$. The *compact representation* [Choi et al., 1993] instead represents edges between names, in a similar fashion to a *points-to* [Emami et al., 1994] representation.

solution size: As a result of the other parameters, the number of solution sets varies. Analyses may have one solution for the whole program, one for each function, or one per program point.

The major analyses of the era vary in these parameters.

Both Andersen's [1994] and Steensgaard's [1996] algorithms are context- and flow-insensitive, modelling only stack variables. They both use only one solution for the whole program, but differ in their directionality: Andersen's algorithm is inclusion-based; Steensgaard's is unification-based. The result is that Steensgaard's algorithm runs in linear time compared to $O(n^3)$ complexity for Andersen's. Both algorithms' run-times are very short for programs less than 3000 lines, but Andersen's can be 10-100 times slower for large programs (6000-25000 lines) [Shapiro and Horwitz, 1997, Section 3.2]. The precision of the algorithms is also very different, with Andersen's being significantly more precise [Hind and Pioli, 2000, Section 5].

Burke et al.'s [1994] algorithm is flow- and context-insensitive, and requires one analysis set per function. Its major difference from Andersen's algorithm is that it is interprocedural. This makes the analysis more accurate because it does not consider unreachable functions, and uses a variable's scope in its analysis. However, this appears to make little difference to the precision in practice, nor does it greatly affect the algorithm's run-time [Hind and Pioli, 2000, Sections 5.1 and 5.6].

Choi et al.'s [1993] algorithm is flow- and context-sensitive and requires two solution sets per program point (control-flow graph node). However, it is only slightly more precise than Burke et al.'s algorithm, but an average of 2.5 times slower [Hind and Pioli, 2000, Section 5]. Its representation is *compact pairs*, which are more compact than *alias pairs*. Compact pairs only represent basic aliases, deriving further alias relationships through transitivity and commutitivity. For example, the pairs ⟨`*a`, `b`⟩ (`*a` aliases `b`) indicates that `a` points-to `b`. Combined with the pair ⟨`*b`, `c`⟩, we could derive the set ⟨`**a`, `c`⟩. This is represented explicitly using alias pairs, but compact pairs omits this set, instead representing it implicitly.

Points-to analysis [Emami et al., 1994] differs from these algorithms in that it is fully context-sensitive, and that it only attempts to model stack locations. It models all heap allocations with a single heap node. Its representation is a *points-to graph* which is similar to *compact pairs* [Choi et al., 2003]. It also adds the idea of *certainty*—whether an alias is *definite* or *possible*. Unfortunately, it does not seem that *points-to* analysis has been compared directly to other major analyses. However, the techniques of the analysis have certainly been exported.

Simpler Forms of Alias Analysis

Address-taken alias analysis keeps all stack-locations which have their address taken (in the sense of the C code: `p = &x;`) in a single set. The set also contains all heap names and global variables. This form of analysis is simple, easy to implement, and very fast, being linear to the size of the program. However, it is significantly less precise than Steensgaard's [1996] analysis.

Type-based alias analysis [Diwan et al., 1998] uses a language's strong typing rules to restrict analyses. It relies on the fact that two memory locations cannot alias if their types are not compatible. Furthermore, two fields of the same object are not the same, even if they have the same type. These rules can be combined with a simple analysis such as Steensgaard's, but the improvement is not significant [Diwan et al., 1998, Section 3.3].

3.2.3 Divergence

In the late 1990s, the interest in pointer analysis coincided with the explosion of interest in Java. The success of Java from its launch in 1995 led to a huge interest in industry, undergraduate teaching and research material. In the field of program analysis, it kick-started a resurgence of JIT compiler technology, and led to the development of advanced garbage collectors. It also led to the development of escape analysis.

Escape Analysis

Escape analysis algorithms attempt to statically detect the lifetime of objects. The simplest use of escape analysis is to allow objects to be allocated on the stack instead of the heap, but common optimizations include synchronization removal and *scalar replacement of aggregates* [Muchnick, 1997].

Escape analysis was first designed to analyse LISP [Park and Goldberg, 1991, 1992, Ruggieri and Murtagh, 1988]. The research was extended to allow optimizations on Java [Blanchet, 1999, Bogda and Hölzle, 1999, Choi et al., 1999, Gay and Steensgaard, 2000, Whaley and Rinard, 1999]. The techniques use for these optimizations were very similar to existing algorithms for alias analysis.

Choi et al. [1999] create a *connection-graph*, which is very similar to the *points-to*

graph [Emami et al., 1994]. Each object in the graph is marked according to how far it escapes from the function in which it is defined:

NoEscape: The object does not escape the function in which it was allocated. This means it is not reachable from an object returned from the function, any parameter to the function, any object in global scope, or any object is a separate thread. However, it may be reachable from callees of its allocation function.

A *NoEscape* object may be allocated on the stack. This is designed to avoid memory allocation and deallocation, by allocating it on the stack using `alloca`, which is automatically freed upon popping the function's stack frame. Objects allocated on the stack in this manner will also see improvements in their cache locality. Stack allocation is a prerequisite for *exploding* an object's fields into scalar registers, known as *scalar replacement of aggregates* or *object explosion*.

ArgEscape: An object which *ArgEscape*s may escape its allocating function, but it never escapes the thread in which it was allocated. This allows synchronization optimizations to be performed on the object. The simplest optimization is to remove they synchronization header from the allocated object, reducing the run-time memory usage. However, in Java it is also possible to remove calls to synchronization primitives, and to replace standard strings and collections with lock-free versions.

In research which performed both stack allocation and synchronization optimizations, the synchronization optimizations were typically much more successful [Whaley and Rinard, 1999, Sections 8.3 and 8.4] and useful [Choi et al., 1999, Section 8.3].

GlobalEscape: Objects which escape globally are the most conservative case, and are not suitable for optimization.

Although the main optimizations allowed by escape analysis are stack allocation and synchronization removal, it can also be used for *pre-tenuring* in a generational garbage collector [Jump and Hardekopf, 2003]. This allocates objects with long-lifetimes in the older generations of the garbage collector, as it is expected that the objects will otherwise need to be moved there from a younger generation.

Despite the early promise of escape analysis, interest died away relatively quickly, for a combination of reasons. As the major application of escape analysis was optimization—as opposed to program understanding for example—the practicality of the optimizations was paramount. However, escape analysis requires whole-program optimization,

which is made extremely difficult by Java's dynamic class loading[3] [Liang and Bracha, 1998]. In addition, the analyses are typically expensive—Choi et al.'s intraprocedural analysis is $O(N^6)$, though it is expected that it could be reduced to $O(N^3)$ [Choi et al., 2003, Appendix A].

Although escape analysis was expensive, the speedups due to it were not large. Choi et al. [1999] provided both stack allocation and synchronization removal, for a median speed improvement of 7%. More importantly, however, the stack allocation benefits which were measured in early analyses were predicated on relatively poor memory management systems. Objects could be allocated on the stack using `alloca`, a cheap operation, and were deallocated for free. However, copying garbage collectors achieve this speed for many more objects: all allocations use a *bump*, with the same cost as an `alloca`, and objects which die quickly and are not copied are deallocated for free. Guyer et al. [2006] note that "it is unlikely that any technique can beat the performance of copying generational collection on short-lived objects". Moreover, stack allocated objects risk increasing the lifetime of an object, actually increasing the memory usage of a program.

3.2.4 Scalability

Modern alias analysis techniques have been focussed on the speed and scalability of alias analysis. The aim of analysing millions of lines in a reasonable amount of time and space has long been an important goal. Recent research has achieved many important goals in this area.

One of the most important designs in alias analysis is that of Andersen [1994], who used a constraint-based approach for his analysis. A constraint-based approach splits the analysis into two parts: the first generates constraints on pointers in the program, the second applies a constraint solver to generate a solution. Dividing the analysis into two parts has enabled significant improvement to both the speed and the scope of the analysis.

Andersen's analysis uses *inclusion constraints*, which have natural graph representations, but typically did not scale past small- to medium-sized programs. Inclusion constraints are important for other program analyses, notably type inference [Palsberg and Schwartzbach, 1991]. Fähndrich et al. [1998] were the first to realize that graph

[3] Choi et al.'s research [Choi et al., 1999, 2003] was implemented in a *static* compiler for Java [IBM 1997].

optimization techniques could increase the speed of solving inclusion constraints by orders of magnitude. Later work further increased the performance of the techniques [Hardekopf and Lin, 2007, Heintze and Tardieu, 2001, Pearce et al., 2007, Rountev and Chandra, 2000, Su et al., 2000], as well as extending their range to be field-sensitive [Pearce et al., 2007]. These techniques are useful in practice, and are currently implemented in `gcc` [Berlin, 2005].

While speeding up Andersen's analysis is important, so too is extending it to be both flow- and context-sensitive, while remaining useful in practice. Both speed and size considerations are important, and both were largely solved. In considering size, the most important development was the use of *Binary Decision Diagrams* (BDDs) [Bryant, 1992]. BDDs are efficient DAG-based representations of boolean functions. They allow for duplicated solution sets to be represented in significantly less space than a naive implementation. There have been a number of investigations into the use of BDDs for alias analysis, which have pointed to significantly reduced memory usage in practice, for context-sensitive analysis [Berndl et al., 2003, Whaley and Lam, 2004]. Hardekopf and Lin [2009] have extended the use of BDDs to flow-sensitive analysis.

Following the success of inclusion constraints, the Saturn project [Aiken et al., 2007] investigated the use of other constraint solving systems. Hackett and Aiken [2006] use a boolean constraint solver to represent a fully flow- and context-sensitive analysis, which is partially path-sensitive. While they do not directly compare precision or analysis run-time with other scalable approaches, their results indicate that hundreds of thousands of lines of code can be analysed in reasonable time.

3.2.5 Shape Analysis

Shape analysis is a means of detecting the *shape* of recursive data structures. Much of the pre-history of alias analysis comes from attempting to derive information about the shape of structures [Chase et al., 1990, Jones and Muchnick, 1982, Larus and Hilfinger, 1988]. Shape analysis can be considered a more precise form of alias analysis. Shape analyses are able to detect structures such as lists, trees, DAGs or cyclic graphs [Ghiya and Hendren, 1996], and can detect destructive techniques such as reversing a list [Sagiv et al., 1996].

3.3 Static Single Assignment Form

Static single assignment (SSA) form provides an efficient intermediate representation for program analysis [Cytron et al., 1989, 1991]. It was introduced as a way of efficiently representing dataflow analyses.

SSA uses a single key idea: that all variables in the program are renamed so that each variable is assigned a value at a single unique statement in the program. From this key idea, SSA is able to provide a number of advantages over other techniques of dataflow analyses:

Factored use-def chain: With a def-use chain, dataflow results may be propagated directly from the assignment of a variable to all of its uses. However, a def-use chain requires an edge from each definition to each use, which may be expensive when a program has many definitions and uses of the same variable. In practice, this occurs in the presence of `switch`-statements. SSA *factors* the def-use chain over a *ϕ-node*,[4] avoiding this pathological case.

Flow-sensitivity: A flow-insensitive algorithm performed on an SSA form is much more precise than if SSA form were not used. The flow-insensitive problem of multiple definitions to the same variable is solved by the single assignment property. The allows a flow-insensitive algorithm to approach the precision of a flow-sensitive algorithm.

Memory usage: Without SSA form, an analysis must store information for every variable at every program point. SSA form allows a *sparse* analysis, where an analysis must store information only for every assignment in the program. With a unique version per assignment, the memory usage of storing the results of an analysis can be considerably lower than using bit-vector or set-based approaches.

Research on SSA form progressed quickly initially, with many important algorithms being adapted to the SSA form, including constant propagation and dead-code elimination [Wegman and Zadeck, 1991], variable equality [Alpern et al., 1988] and value numbering [Rosen et al., 1988]. Many other important optimizations have also been adapted, including partial-redundancy elimination [Chow et al., 1997, Kennedy et al., 1999], type inference [Lenart et al., 2000] and value-range propagation [Patterson, 1995]. Important research was also performed to provide more precise or efficient

[4] '*phi*'

SSA construction algorithms [Aycock and Horspool, 2000, Brandis and Mössenböck, 1994, Briggs et al., 1998, Das and Ramakrishna, 2005, Sreedhar and Gao, 1995], and a means to iteratively reconstruct the SSA form after program transformation [Choi et al., 1996].

3.3.1 SSA Form and Alias Analysis

SSA form is principally designed to optimize scalar variables. As a result, there have been many approaches to adapting SSA form for the aliasing which can be found in many C and C++ programs.

In the original SSA research [Cytron et al., 1991, Section 3.1], aliasing was already considered, and arrays, structures, and implicit references are all discussed. The techniques they provide are simple extensions to a scalar SSA form. Pointers to arrays or structures are treated as scalar values – this allows Java programs to be converted to SSA form without extension. Arrays are treated as a single scalar value. This safely represents an array in SSA form, but the representation is opaque, and it is not possible to analyse through arrays with this scheme. Their scheme provides *Update* and *Access* operators to ensure that arrays remain live through multiple assignments. Structures are treated the same as arrays, with the exception that it may be possible to treat each field of a structure as a separate scalar value.

In order to handle aliasing, they suggest modelling the entire heap as an single node, because it should still be possible to optimize scalar code efficiently with this simplification. To actually model pointers, they add new operators: *MustMod*, *MayMod* and *MayUse*. They add these operators to their IR as assignment statements before and after indirect assignments. Naturally, this mirrors work on alias analysis (e.g [Emami et al., 1994]).

This technique can represent alias information precisely and accurately, but can lead to memory explosion if aliases are represented more finely than their approach of using a single node to represent the entire heap. For example, Landi and Ryder [1992] report an average of 675 may-alias relations per node. Representing this information in the manner suggested by Cytron et al. [1991] would be impractical.

SSA arrived during the ascendency period of the development of alias analysis. By the time that powerful alias analysis techniques were created, SSA had already established itself as an important framework for program analysis. With more powerful analyses

$x = z;$	$\mu(a_1);$ $*x_5 = z_2;$ $y_2 = \chi(y_1);$

Listing 3.1: *Example of HSSA form. This assumes that z may alias a, and that x may alias y.*

and precise results, a natural extension to SSA was to allow it model aliases more precisely, and to control the memory explosion when this occurred. While early work [Cytron et al., 1991] was aware of means of modelling references more finely [Chase et al., 1990], they do not discuss the potential explosion in memory use. As a result, other techniques techniques were developed to integrate this precise alias information into the SSA form.

HSSA

The most powerful and general of the techniques for combining SSA with aliasing information is *Hashed SSA* (HSSA) [Chow et al., 1996]. HSSA not only models aliasing information in SSA form, it also provided techniques for managing the memory usage.

HSSA models may-uses and may-definitions by annotating statements with μ and χ operations respectively. Figure 3.1 shows a C indirect assignment in HSSA form. The example assumes that z may alias a, and x may alias y. The indirect assignment to x may-defines y, leading to the assignment to y_2 from an unknown value. The assignment uses z_2, which may use a_1. Uses via μ operations semantically occur before a statement, and definitions via χ operations semantically occur after a statement.

Naturally, with very large alias sets, the number of μ and χ nodes explodes. HSSA provides a number of techniques to handle this explosion. *Zero versioning* uses a special variable version for variable occurrence which are not *real*. A *real* occurrence is any which does not arise from χ, μ or ϕ nodes. Zero versioning condenses chained χ nodes to use a single version (0), at a slight cost to precision. Zero versions are included as soon as HSSA form is built, and reduces the number of χ nodes which are chained.

HSSA also introduces virtual variables. A virtual variable can represent any set of variables, and is intended to reduce the large sets of χ nodes into a single node. The choice of what to represent with a virtual variable remains in the hands of the analysis author, but a simple choice might be to have a single virtual variable represent the

contents of an alias set.

Using these techniques, HSSA is intended to make SSA form with aliasing manageable. As a result, it has been adopted in a number of compilers, such as the Scale compiler [Chowdhury et al., 2004], `gcc` [Novillo, 2007], the WOPT from the SGI Open64 compiler [Kennedy et al., 1999].

Unfortunately, HSSA comes with a cost. Since χ operations are not real, they must be dropped when SSA form is removed. This requires that variables in SSA form are not propagated across a variable's live range, restricting analyses such as copy-propagation. In `gcc`, this is somewhat mitigated by keeping scalar types in SSA, and only using HSSA for indirect memory operations [Novillo, 2007].

Other SSA Extensions with Alias Information

HSSA is not the only kind of SSA form supporting alias information.

Extended SSA Numbering [Lapkowski and Hendren, 1998] attempts to bring the benefits of SSA to languages with multi-level pointers. The cost of their approach is that not all variables have a unique definition, meaning that important SSA properties are not supported.

Another approach to managing alias information in SSA form is to incrementally improve the precision of may-alias information as it is needed [Cytron and Gershbein, 1993]. As the amount of alias information available is often larger than the amount required to perform an optimization, it is possible to increase the precision of the results lazily when more information is required. This means that alias information which is never used is not calculated. Unfortunately, this requires rebuilding the SSA form each time extra precision is required, although they claim that this is not a problem in general.

It seems that these techniques have not been used in practice, however.

Other Related SSA Extensions

Array-SSA [Knobe and Sarkar, 1998] provides a means of modelling assignments to arrays and structures in SSA form. It provides ϕ nodes to model definitions and uses of array values, and uses them to avoid data dependences during automatic parallelization.

Although it models dependences between arrays and structures, it is not a general means to support aliasing information.

Heap-SSA [Fink et al., 2000] is an extension of Array-SSA for strongly typed languages, allowing elimination of dead stores. It relies on global value numbering and strong type information, instead of alias analysis, to differentiate between different values. As it relies on strong type information and does not support alias analysis, I do not discuss it further.

Although HSSA mitigates the memory explosion somewhat, it is not sufficient for very large programs. To counteract this, `gcc` uses a technique called *Memory-SSA* [Novillo, 2007]. This technique allows the size of the HSSA alias sets to gracefully fall back to a less-precise form in the presence of a memory explosion.

Despite the promise of Memory SSA, however, its benefits did not materialize [Guenther, 2009]. For large programs (greater than one million lines of code), even Memory SSA does not scale in its memory usage. As a result, representing alias information directly in SSA still appears to be an open problem. Modern versions of `gcc` keep only scalar variables in SSA, and use an *alias oracle* to disambiguate between memory locations [Guenther, 2009].

3.4 Type Analysis

Scripting languages are dynamically typed, meaning that the types of variables, fields and array offsets are not provided in the program text. As a result, it is important to perform type analyses on the program in order to regain the information that is lost by virtue of latent typing.

Type analysis is a broad area, encompassing many techniques and algorithms designed to extract type information from a program text. It is closely related to type feedback techniques described in Section 2.2. Static type analyses can be broadly divided into three categories:

1. analyses to support devirtualization in statically typed object-oriented languages,
2. type-inference of dynamic languages,
3. type-inference of statically typed functional languages.

3.4.1 Devirtualization Support

Method dispatch in statically typed object-oriented languages such as Java and C++ is typically implemented using *virtual dispatch tables* (*vtable*). A vtable is a table of method pointers. At compile-time, each named method is associated with a table offset. At run-time, the method at that offset is dispatched to.

As the static type of the object must be known, it is possible to know the correct offset to which to dispatch. However, the run-time type of the object may be any subclass of the static type, and the method dispatched to may be overridden in the subclass. As this *dynamic dispatch* is costly, devirtualization techniques have been developed to optimize it to static dispatch. Turning a virtual method call into a static method call also enables inlining and more accurate callgraphs, which also enable other optimizations.

Each devirtualization algorithm associates a set of class names with objects in the program. By using more sets with smaller granularity, the algorithms can achieve greater precision at the cost of analysis time. Class Hierarchy Analysis (CHA) [Dean et al., 1995] analyses all callable methods using a single set. Starting at the `main` method, it searches a method for the static classes of the receiver of a method invocations, and adds all subtypes of the receiver to its set. It then recurses into reachable methods to continue the analysis.

Rapid Type Analysis (RTA) [Bacon and Sweeney, 1996] prunes the set using information about instantiated classes. During analysis, it stores the set of instantiated classes. At method invocation time, it reasons that only already instantiated classes may the run-time type of the receiver. Using this extra information, RTA can be significantly more precise than CHA.

Tip and Palsberg [2000] develop a series of analyses named CTA, MTA, FTA and XTA. XTA uses a set of classes for each method and field of a class, instead of a single set for the whole analysis. The sets contain classes which have been instantiated and are subtypes of their static types. The sets are propagated to callees via parameters—that is, the parameters' subtypes which have been instantiated initialize the set in the callee—and back to callers via return types. Tip and Palsberg [2000] report this analysis allows an average of 12.5% of virtual methods can be devirtualized, although the analysis runs over 8x slower than CHA. The other algorithms, CTA, MTA and FTA, investigate the design space between RTA and XTA. Tip and Palsberg [2000] report that XTA offers the greatest trade-off between precision and run-time.

Lenart et al. [2000] use an SSA-based propagation algorithm, based on constant propagation [Wegman and Zadeck, 1991]. It uses a more traditional dataflow style, propagating instantiated types to call-sites. It also allows constant-propagation information to make the results more precise, as discussed in Section 3.5. Unfortunately, they do not evaluate its precision or the speed of the algorithm.

3.4.2 Type-inference

The last section described a way of discovering more precise type information in statically typed object-oriented languages. For languages which do not require type annotations, *type-inference* algorithms analyse programs to compute this type information. This section discusses existing work on type inference. Section 2.2 discusses the related area of optimizations based on type-feedback at run-time.

Dynamic Languages

In order to optimize the code generated by a compiler or JIT compiler, it is useful to infer a run-time value's type. Similarly, tools such as IDEs, code browsers and bug finders benefit from knowing the type associated with memory locations in a program.

A type in a dynamically typed language may refer either to the run-time class or type-tag of a value, or to its structure or interface, as is the case in a duck-typed language (see Section 2.1.3). Type inference algorithms may detect either or both of these meanings of *type*.

A powerful type-inference system known as the *Cartesian Product Algorithm* (CPA) was developed for the analysis of SELF [Agesen, 1995]. It is based on a previous algorithm by Palsberg and Schwartzbach [1991].

Palsberg and Schwartzbach's [1991] algorithm is a constraint-based algorithm, in the same way as Andersen's [1994]. It begins by allocating type variables for expressions and lvalues in the program. It seeds these type variables with literals from the program. For example, for the statement `$x = 5`, the type variable for `$x` is seeded ⟨`int`⟩. Constraints are then established based on the assignments and methods calls in the program. The constraint graph is then solved, establishing a solution set of types for each type variable in the program.

The main problem solved by CPA which is not solved by Palsberg and Schwartzbach [1991] is that the algorithm is imprecise for methods called with multiple parameter types. If an identity function is called with a parameter typed with the set ⟨`int`, `real`⟩, the return set is ⟨`int`, `real`⟩. The CPA algorithm will instead split the solution set into two, with the types ⟨`int`⟩ and ⟨`real`⟩. This will give a more precise context-sensitive result of ⟨`int`⟩ and ⟨`real`⟩, respectively.

CPA is very similar to Andersen's [1994] analysis, in that both use inclusion constraints and solvers to achieve their solution. In fact, many of the insights regarding the speed and scalability of constraint-based alias analysis came via the study of type-inference.

We discuss the application of CPA to dynamic scripting languages in Section 3.6.1. Research on the Self compiler [Agesen and Hölzle, 1995] indicates that using type-feedback for optimization is equally effective as performing static type-inference.

Statically Typed Functional Languages

In order to detect a class of errors in programs, statically typed functional languages such as Haskell, Clean or OCaml have complex type systems, typically based on the Hindley-Milner type system [Milner, 1978]. The Damas-Milner type-inference algorithm [Damas and Milner, 1982] is used to perform type inference on the Hindley-Milner type system. Dynamic languages do not use type systems based on Hindley-Milner. While it would be possible to retrofit Hindley-Milner to a dynamic language such as PHP, it would be non-trivial due to features such as those described in Section 6.2.5. As such, we do not consider these type inference algorithms, and do not discuss them further.

3.5 Combined Analysis

Compilers often suffer from a *phase ordering* problem. Given two optimizations A and B, A may create opportunities for B, which causes more opportunities for A. This is often solved by iteration: A and B are executed alternately until the program reaches a fixpoint, and there are no more optimizations to be performed.

A common example is the combination of constant-propagation and unreachable-code

```
1  int x = 1;
2
3  do
4  {
5      if (x != 1)
6          x = 2;
7  }
8  while (...);
9
10 return x;
```

Listing 3.2: *Program which can be better optimized using a combined analysis than iterating over separate analyses. This example is taken from Figure 8 in Click and Cooper [1995].*

elimination.[5] Constant-propagation may resolve known branches, creating opportunities for unreachable-code elimination. The eliminated code may lead to a more precise constant-propagation, creating further unreachable code.

However, there are benefits in combining these analyses. Click and Cooper [1995, Section 6.2] showed that an algorithm which combines unreachable-code elimination and constant-propagation is strictly more powerful than iterating over the two. Consider the example in Listing 3.2. On line 5, unreachable-code elimination will remove the statement on line 6. Constant-propagation does not then consider it, and the value of `i` is known to be `1` at line 10.

Conditional constant propagation (CCP) was discovered by Wegbreit [1975], but made popular through Sparse CCP (SCCP) [Wegman and Zadeck, 1991], which combines CCP with SSA. By using SSA's factored def-use chains, SCCP avoids pessimistic cases. The CCP algorithm is expensive, requiring one lattice per variable per program point. SCCP is less expensive, taking advantage of the sparse representation from SSA.

SCCP uses a simple algorithm. Beginning with the entry node, it analyses statements in a function using a *symbolic execution*. At each assignment of a constant to a variable, it stores the value of the variable. Operations between variables, such as addition, are folded when all operands are known by the analysis. At each branch statement, the branch condition is evaluated. If it is known, only the known direction is deemed to be *executable*. Only executable branches are followed. If the branch condition has an unknown value, both branch targets are executable.

During the analysis, any variables which are defined have all of their uses added to a

[5] Unreachable code is sometimes known as *dead-code elimination* (DCE). However, DCE sometimes refers to removing useless or redundant code, so we do not use the term here.

worklist.[6] The algorithm continues until the worklist is emptied. The algorithm is optimistic, in that it is not sound to end the optimization before it has completed.

A number of other analyses have been created using the SCCP algorithm, including value-range propagation [Patterson, 1995] and type-inference algorithms [Lenart et al., 2000]. Click and Cooper [1995] provide a framework for combining algorithms, combining SCCP with value numbering, and reasoning about the combined analyses.

Lerner et al. [2002] created a technique which allows the automatic combination of separately described algorithms. This technique is at least five times faster than iterating over the separately described algorithm, and has an overhead of less than 20% compared to hand-optimized combined analyses.

The `gcc` compiler uses the SCCP algorithm as a general propagation algorithm [Novillo, 2005], while the Glasgow Haskell Compiler (`ghc`) uses Lerner et al.'s [2002] algorithm [Ramsey et al., 2009]. In Section 4.6.4, I show a novel SSA construction algorithm for use simultaneously with an SCCP algorithm.

3.5.1 Combined Constant-propagation and Alias Analysis

Pioli et al. [1999] describe how to combine alias analysis with constant-propagation. This allows a more precise alias analysis, because unreachable paths will not be considered. Our algorithm follows a similar structure to Pioli et al., as described in Section 6.5, so we look at it in more detail here.

Pioli et al.'s algorithm is a combination of the algorithms of Burke et al. [1994] and Wegbreit [1975]. We give an overview of the algorithm:

- The analysis performs a symbolic execution of a program. Beginning at the first statement in the `main` method, the analysis is performed one statement at a time. At every statement, both the alias analysis and constant propagation solutions are updated.
- Pioli et al.'s analysis is flow-sensitive and context-insensitive. It uses a worklist algorithm to track the basic blocks in a function, processing them in topological order. It is a *conditional* analysis, meaning it attempts to resolve branch state-

[6] This is a slightly simplified explanation. The algorithm actually uses two worklists, one for SSA def-use edges, and one for CFG edges.

ments immediately using the constant propagation. This results in an optimistic analysis, which must complete in order for its results to be correct.

- At a flow-control join point, the results in the predecessor blocks are merged. When the analysis changes a basic block's solution, that block's successors are added to the worklist. Both of a branch block's successors are added, unless the branch has been conditionally resolved. When the analysis results reach a fixpoint, the worklist will empty.

- Upon reaching a method invocation, the analysis on the current method pauses. Current analysis results are copied to the callee, termed a *forward-bind*. The analysis then continues processing using a new worklist, beginning at the entry block of the invoked function. The callgraph is built lazily, but multiple functions may be invoked from the same location, due to function pointers. In this case, all possible callees are analysed.

- When a method has been fully analysed, the analysis results are copied back to the caller, termed a *backward-bind*. If there are multiple possible receivers, their results are merged. The caller's worklist then continues.

3.6 Static Analyses of Dynamic Scripting Languages

This section discusses existing research into static analysis of scripting languages. There exists only a small amount of research into the area, so this is discussed is some detail. Section 2.2 discusses research into scripting language implementations, which is orthogonal to this section.

3.6.1 Type Analysis

The most important and well-studied static analysis for scripting languages is type inference. Analysis algorithms used for scripting languages vary in their complexity and precision.

Aggressive Type Inference [Aycock, 2000] is a very simple technique for analysing types of variables in Python. It uses only nominal types, and is not flow- or context-sensitive. Its effectiveness is not reported.

Adaptations of Earlier Analyses

Two important techniques in the analysis of dynamic languages were CPA [Agesen, 1995] and iterative type analysis [Palsberg and Schwartzbach, 1991]. These are discussed in Sections 3.4.2 and 3.4.2, respectively. Naturally, dynamic scripting language analyses have been developed by adapting these techniques.

Iterative type analysis was used as the basis of Cannon's [2005] analysis. He explored whether a type inference algorithm could be performed on Python which led to a performance improvement of 5% or more in the CPython interpreter,[7] without making changes to the language or bytecode compiler. As such, it only attempted to infer types locally. After analysis it added type specific bytecodes to take advantage of the inferred information. This is similar to the run-time component in *Dynamic Interpretation* [Williams, 2009]. However, Cannon reported a speedup of 1% on average, which he felt did not justify the complexity of the approach.

A modified CPA algorithm was used in Starkiller [Salib, 2004], a type-inference tool for Python. The algorithm was modified to handle data polymorphism which is not well handled by CPA. However, it does not handle a number of advanced Python features such as exceptions, iterators or generators. Unfortunately, Salib does not report on the power or precision of the type inference, instead benchmarking a short simple benchmark involving a factorial calculation.

Shed-skin [Dufour, 2006], a Python-to-C++ compiler, uses both iterative type analysis and CPA. It analyses Python programs, and generates C++ classes from the Python classes in the program. As C++ is a static language, it is not straightforward to map Python classes to C++ classes. Shed-skin introduces the idea of *class splitting* in which variables with more than one static types are split into multiple variables, one for each static type. With these extensions, their compiled code runs on average 45 times faster than equivalent CPython code in some cases. However, they are not able to deal with many features of Python. In particular, many Python programs cannot be statically typed using their algorithm, and so cannot be compiled by Shed-skin.

Advanced Type Analyses

Recent type analysis algorithms for scripting languages have modelled significantly larger portions of the languages they analyse, using sophisticated techniques.

[7] The CPython interpreter is the canonical implementation of Python.

Diamondback Ruby (DRuby) [Furr et al., 2009] is a static type inferencer for Ruby, aimed at statically detecting run-time method invocation errors. It provides a constraint-based inference algorithm, along with user type annotations. Unlike analyses for Python, they model the Ruby language in significant detail. As well as nominal types, they also model structural typing, representing Ruby's duck-typing system. They effectively handle Ruby's inheritance with intersection types, and dynamic typing with union types. They model arrays and other generic data structures using parametric polymorphism.

One of the most important problems in Ruby in the use of run-time code generation. Many libraries use calls to the `eval` function, which have side-effects including defining new methods, and adding them to user objects passed into the libraries. *P*Ruby [Furr et al., 2009], an extension to DRuby, handles this through the use of feedback-directed analysis. They profile the program to store the strings run using `eval` at run-time, then perform the static analysis using the results. This enables them to analyse nearly the full program. They report that most Ruby programs they have tested only use a small number of strings in their `eval` statements. In addition, the profiling support allows them to eliminate a number of false positives in their underlying static analyser.

Jensen et al. [2009] developed a type-inference algorithm for JavaScript. Their major contribution is modelling the entire Javascript language in a single lattice-based abstract interpretation. Their lattice models strings, integers, types, Javascript's protocol-based inheritance, object fields, undefined variables and fields, and Javascript-specific attributes of fields. Notably it also includes points-to information to model the fluid structure of Javascript objects. It is also very precise, being both flow- and context-sensitive, and it is capable of verifying the absence of a number of errors. We discuss this work in further detail in Section 6.7.

3.6.2 Alias Analysis

Pixy [Jovanovic, 2007, Jovanovic et al., 2006,] is the state-of-the-art in alias analysis for PHP. It is designed as a static taint analysis for PHP 4, aimed at detecting *cross-site-scripting* (XSS) and vulnerabilities. Its major contribution is that it models references between variables. However, it does not model a great deal of the PHP language. Although it is interprocedural, it does not model object methods[8] and therefore does not types. It also does not model fields or object members. We discuss Pixy in more

[8] Pixy was designed for PHP 4 which is not object-oriented.

detail in Section 6.3.3.

Points-to analyses have also been built for Javascript. Jang and Choe [2009] present an alias analysis for a subset of Javascript, based on Andersen's analysis. It unifies the modelling of objects and tables, and models the unknown field of an object. In their type analysis for Javascript, Jensen et al. [2009] also analyse the points-to relationships of objects, for the full Javascript language. Javascript has a relatively simple points-to model, and their analysis reflects this. It models object fields and what other named objects they may point to.

3.6.3 Other Analysis

String manipulation is one of the most important facilities provided by a scripting language. A number of analyses are designed to model the string values in a program. Minamide [2005] created a static string analyser for PHP. It was aimed creating a link between the inputs and outputs of a PHP program, in order to detect XSS vulnerabilities. It could also be used to statically validate the HTML output of a PHP program. The analysis was based on string analysers for Java [Christensen et al., 2003]. It modelled strings as they flowed through the program, including complex string transformations using regular expressions. Wassermann and Su [2007] extended this technique to model SQL injection vulnerabilities. They used to same technique to model PHP's dynamic `include` statements.

Ben Asher and Rotem [2009] investigate the effect of loop unrolling and method inlining in a bytecode optimizer for Python. They use traditional dataflow analyses such as copy propagation, common sub-expression elimination and loop-invariant code motion to optimize Python programs. Their key insight is that these optimizations are significantly more effective as a result of inlining or loop unrolling. Unlike many scripting language analyses, they deliberately do not use type inference in their analysis, and do not evaluate optimizations based on types.

We discuss other static analyses of PHP [Huang et al., 2004, Xie and Aiken, 2006] in Section 6.3. Scripting language implementations, including run-time analyses, are discussed in Chapter 2.

3.7 Summary

This chapter discusses important program analyses, including alias-analysis, SSA form, and type-inference. This provides important background information for Chapters 4 and 6. It also describes how static analyses have been applied in scripting languages, including how the design of scripting language compilers have affected the analysis.

In the next chapter, we turn our attention to the design of our compiler, `phc`. We see how program analyses and ahead-of-time compilation interact for scripting languages, and how a compiler must be specially designed for the task of compiling scripting languages.

Design of the phc Compiler

In producing the research presented in this dissertation, I built most of the middle- and back-ends of the phc compiler. The phc compiler is an open-source, ahead-of-time compiler for PHP, written in C++. Although phc is structured as a traditional multi-pass compiler, many of its design decisions are based around features of the PHP language.

This chapter describes how phc was designed around the PHP language. It provides an experience report of the challenges of building a compiler for a dynamic language. It does not provide comprehensive overview of the structure of phc, but rather highlights design decisions caused by PHP's unusual and dynamic nature.

Section 4.1 presents a short overview of phc. Section 4.2 discusses the design of phc's intermediate representations. Section 4.3 discussed the challenges in compiling a language that has grown up in an interpreted environment. Section 4.4 presents a description of phc's passes. An overview of the phc's interactions with the PHP system is presented in Section 4.5. Section 4.6 outlines the phc optimization framework.

4.1 Overview

The phc compiler follows the traditional design of ahead-of-time compilers, parsing a PHP program, translating it through a number of Intermediate Representations (IRs), and generating an executable. Its structure is shown in Figure 4.1. The phc compiler:

- parses PHP source code into an *Abstract Syntax Tree* (AST),
- simplifies the AST into a *High-level Intermediate Representation* (HIR), a subset of the PHP language based on *three-address code* (TAC),
- simplifies the HIR into a *Medium-level Intermediate Representation* (MIR), introducing constructs which are not part of PHP,
- performs dataflow optimizations on the MIR,

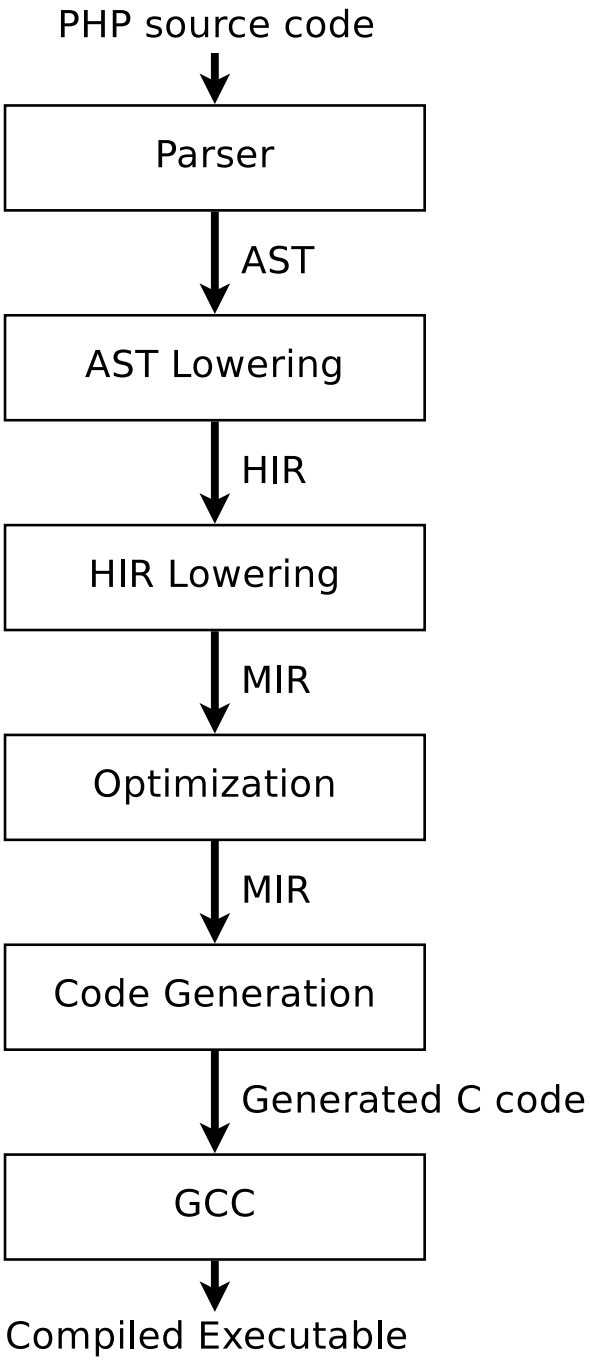


Figure 4.1: *Overview of the structure of phc*

- generates C code from the MIR (see Section 5.4.4),
- compiles the C code using `gcc`, producing an executable.

In discussing `phc`, we wish to focus on the portions that were built to carry out the research in this dissertation. We build on the existing `phc` front-end [de Vries and Gilbert, 2007]. In addition to the content of this chapter, `phc`'s code generation and link to the PHP system are discussed in Chapter 5, and the design of the `phc` optimization framework is described in Chapter 6.

4.2 Intermediate Representations

There are three Intermediate Representations (IRs) [Muchnick, 1997] used in `phc`, as described in the previous section: the AST, the HIR and the MIR. Each IR is defined using `maketea` [de Vries and Gilbert, 2008], a tool for building C++ class hierarchies from IR definitions, along with visitor and transformation APIs. These APIs are used by most of the passes described in Section 4.4. Each IR is defined in `maketea` format in Appendix A.

4.2.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST) closely represents the syntax of PHP programs. It is generated directly by the parser, and abstracts only syntactic elements of the source program, such as commas and semi-colons. As the AST is such a close representation of a PHP program, we are able to convert (or '*unparse*') the AST back into source code. This allows `phc` to be used as a source-to-source compiler, where both the initial and target languages are PHP.

This is extremely useful in testing the compiler. The behaviour of a PHP program is defined by running it with the PHP system's interpreter. We can tell that `phc`'s passes operate correctly if the new program behaves identically to the original. This allows the compiler to be tested easily.

Additionally, we can automatically create new test files by writing plugins to `phc` which transform existing test files into new test files. As `phc` functions as a source-to-source compiler, it is easy to create plugins which perform small or large transformations on

a PHP file. For example, we test our implementation of the PHP **eval** statement by converting every second statement of a PHP program to use an **eval** statement instead of being executed directly.

A PHP program is kept in the AST during conversion to *three-address-code*, at which time it is equivalent to the *High-level Intermediate Representation*.

4.2.2 High-level Intermediate Representation

The High-level Intermediate Representation (HIR) is intended to represent the smallest subset of the PHP language possible, without introducing features to the IR which are not valid PHP. It removes redundant constructs from the AST, and where possible replaces complex nested expressions with simple three-address-code (TAC) [Muchnick, 1997] statements.

The HIR exists primarily as an intermediate step between the AST and the *Medium-level Intermediate Representation*. Using a separate IR allows us to take advantage of C++'s type-safety, and provides a more convenient representation than the AST on which to perform some transformations. It is the final step at which the entire program is valid PHP and can be converted to PHP source code, again useful for testing purposes.

4.2.3 Medium-level Intermediate Representation

The HIR is the most simplified for of the PHP language which is still valid PHP. However, there are a number of further simplifications which can be made, which are not valid PHP. These simplifications are made in the Medium-level Intermediate Representation (MIR). The simplest example of such a construct is the presence of `goto`, which is not a part of PHP.[1]

4.2.4 Challenges in Intermediate Representation Design

This section describes a number of the challenges in creating useful IRs in which to work on PHP.

[1] Actually, `goto` was introduced in PHP 5.3, but most versions of PHP do not have `goto`.

```
if (...)
    class X { ... }
else
    class X { ... }
```

(a)

```
class X0 { ... }
class X1 { ... }

if (...)
    class_alias ("X", "X0");
else
    class_alias ("X", "X1");
```

(b)

Listing 4.1: *Dynamic class declaration at run-time. The definition of* `X` *will depend on run-time control-flow.*

```
$x =& $a[0][1][2];
```

(a)

```
$T1 =& $a[0];
$T2 =& $T1[1];
$x =& $T2[2];
```

(b)

Listing 4.2: *An array assignment, by reference, before and after conversion to three-address code*

Dynamic Declarations

In static object-oriented languages, declarations such as methods, classes and interfaces are provided by the programmer at compile-time. However, PHP allows these declarations at run-time, and the particular class[2] declared may depend on the control-flow of the program. Typically, this occurs because of conditional source inclusion. However, it also also possible for the programmer to write different versions of the same class depending on some run-time control-flow, shown in Listing 4.1a.

As such, these declarations are all statements in the AST and the HIR. However, it greatly simplifies code generation if all declarations are moved to the top-level in the MIR. As a result, we move dynamic declarations to the top-level, and name them uniquely. They are then replaced with `class_alias` statements, as shown in Listing 4.1b. At their original position, an alias is created at run-time between the new and original names. This significantly simplifies the code generation of dynamic declarations.

```
foo ($a[0][1]);
```

(a)

```
if (param_is_ref (foo, 0)) {
    $T1 =& $a[0];
    $T2 =& $T1[1];
} else {
    $T1 =& $a[0];
    $T2 =& $T1[1];
}
foo ($T2);
```

(b)

Listing 4.3: *A method invocation, by reference, before and after conversion to three-address code*

Three-address Code and Assignment-by-reference

It is straightforward to convert an assignment into three-address-code (TAC), as shown in the example in Listing 4.2. The conversion of assignments to TAC is syntactic, based on the presence or absence of reference assignment (`&=`). Reference assignment, indicated by `&=`, makes two run-time names refer to the same run-time memory location.

However, consider the example in Listing 4.3a. As PHP functions are not required to be statically declared, we do not always know the signature of the function being called. In addition, PHP allows polymorphic method calls, and different versions of a method may have differing signatures. As such,we cannot convert the array expression in Listing 4.3a to TAC, as we do not know if the parameter to `foo` should be passed by value or reference. As such we must add a `param_is_ref` construct to the MIR, which evaluates at run-time to a boolean indicating whether the n^{th} formal parameter of the function to be called, requires passing by reference. This allows parameters to be converted to TAC.

As the `param_is_ref` statement is not valid PHP, we are not able to include it in the HIR. As a result, method invocations are *not* converted to TAC before the MIR.

`unset` Statements

An **`unset`** statement clears a particular variable, array entry or object field. After a name is unset, it is exactly as if it had never been initialized. For example, accessing a variable after it is unset evaluates to `NULL`, in the same manner as an undefined variable.

[2] We'll continue using class declarations in our example, but the same is true of methods and interfaces.

```
foreach($arr as $key => $val)
{
    ...
}
```

(a)

```
  foreach_reset($arr, $iterator);

lCheck:
  $THK = foreach_has_key($arr, $iterator);
  if (!$THK) goto lFinalize;

  // Perform body
  $key = foreach_get_key($arr, $iterator);
  $val = foreach_get_val($arr, $iterator);
    ...

  // Increment iterator
  foreach_next($arr, $iterator);
  goto lCheck;

lFinalize:
  foreach_end($arr, $iterator);
```

(b)

Listing 4.4: *A* `foreach`*-loop in the AST and MIR*

Similarly, unsetting the field of an object or array removes that field.

However, it is not possible to convert `unset` statements to TAC in an IR. If this was converted to TAC using assignments-by-copy, it would not unset the originally intended memory location. In the assignments were by reference, we might actually create new values by conversion to TAC. For example, the statement `unset ($a[0][1]);` would be converted to

```
$T =& $a[0];
unset ($T[1]);
```

which might create a value in `$T`. As such, `unset` statements are never converted into TAC. The `isset` expression, which checks whether a variable heas been set, is also treated in this manner.

`foreach` Statements

A `foreach` statement is a loop which iterates through all keys and values of an array or object. All `foreach` statements are kept in the HIR, as they cannot be simplified without introducing non-PHP constructs. To support them in the MIR, we introduce six constructs, which resemble how `foreach` statements are implemented in C. A

typical **foreach** statement is shown in Figure 4.4, in its AST/HIR and MIR forms. In PHP, arrays are iterated through using iterators, which are manipulated by the **foreach** statements.

A **foreach** statement is converted into three statements in the MIR:

- `Foreach_reset`: initializes a **foreach** statement, and point the iterator to the first array element,
- `Foreach_next`: increments the iterator to point to the next array element,
- `Foreach_end`: finalizes a **foreach** statement. This does not do anything in the MIR, but it frees resources in the generated C code.

To access values from **foreach** statements, three expressions are used:

- `Foreach_has_key`: a boolean expression; indicates whether an iterator still points to a valid value,
- `Foreach_get_key`: fetches the key pointed to by the iterator,
- `Foreach_get_value`: fetches the value pointed to by the iterator.

At later stages in compilation, we perform analyses on these statements. For example, during dead-code-elimination, `Foreach_get_value` or `foreach_get_key` statements can be eliminated if they are unused, or the entire loop can be removed if it has no effect. The `phc` code generator also works directly on these constructs.

Loop Design

In the HIR, we attempt to remove **do-while**-, **while**- and **for**-loops, replacing them with a single canonical loop type. This loop is simply called `Loop`, and it loops infinitely, representing **while (true)**.

The CIL compiler and intermediate language [Necula et al., 2002] performs the same simplification. However, CIL uses **goto** statements in their IR, which PHP does not. This allows CIL to cleanly represent all loops with a single construct, while supporting **continue** and **break** statements. Without a **goto** statement in the HIR, we must instead use conditional statements to check the iteration number.

```
for ($i = 0; $i < $N; $i++)
{
    ...
}
```

(a)

```
$i = 0;
$T = True;

while (True)
{
    // Only increment on first iteration.
    if ($T)
        $T = False;
    else
        $i++;

    // Check loop condition
    if (!($i < $N))
        break;

    ...
}
```

(b)

Listing 4.5: *A* `for`*-loop in the AST and HIR.* `$T` *checks the which loop iteration is being executed, avoid a loop increment on the first iteration.*

Listing 4.5 shows a `for`-loop in the AST and the HIR. This is unfortunately both awkward and slow. A better means of handling this might be to check for the presence of `break` or `continue` statements, but that makes the lowering pass more complicated. Instead, we clean this up with an optimization pass, described in Section 4.6.2.

4.3 Compiled and Interpreted Models

The PHP language is defined by the `php` implementation, which is interpreted. Several of PHP's features make more sense in an interpreted model than in a compiled one. As a result, it is useful to either not support, or in some way adapt, these features for a compiled model. We describe these minor variations below:

- A `declare` statement allows the programmer to write a snippet of code which is regularly called by the PHP system's interpreter. This code is executed after a defined number of *interpreter ticks*. When using a compiler instead of an interpreter, it is no longer clear what an *interpreter tick* means. While this could be redefined to mean a certain number of statements, is this really desirable? For example, ticks are generally not called in library code, but only in interpreted

code. The phc compiler allows users to write create libraries by compiling PHP code, instead of writing them in C. In this circumstance, should declare statements be called?

We decided not to support ticks in phc. In choosing not to support this, we also took into account that ticks are very seldom used in real PHP programs, and that the implementation is slightly broken in the PHP system, making this feature unreliable.

- The PHP function debug_zval_count prints the reference count of a value in the PHP system. As the number of references to a value is somewhat arbitrary, different versions of the PHP system may give different results for this function. Similarly, trying to keep the number of references to a value identical to the php implementation would severely constrain the transformations and code generated by phc. As a result, we do not consider the result of debug_zval_count when assessing if a program is correctly implemented by phc.

- In converting the HIR to the MIR, control-flow structures are simplified to using **goto** statements. This means that we do not support an unusual edge case when a loop contains an **eval** statement which calls **break** or **continue**.

- Due to the existence of variable-variables (see Section 5.2.4), every PHP variable name may potentially be used by the program. However, we wish to create temporary variables to support TAC. As such, it is reasonable to claim a prefix for our temporary variables, such as __PHC__. We use the same solution for variable-functions, -methods, -interfaces and -classes.

4.4 Passes

Compilers are traditionally structured as a series of passes, each of which performs a small task. This section provides a description of all of phc's passes, along with the small task which they perform. They are shown to provide a more detailed overview of the compilation process. Figures 4.1–4.6 show the full set of passes which phc performs on a PHP program.

check	Check for invalid PHP statements which do not create syntax errors, such as nested class definitions, or taking a reference to literal values.
incl1	Optionally recursively replace **`include`** statements with the included files.
pretty-print	Optionally print the program source, formatted to a common style.
const-fold	Fold constant expressions.

Table 4.1: *AST passes*

decomment	Remove comments from the AST.
sua	Remove the attributes used to format source code.
ntld	Make a note of top-level declarations, such as methods and classes, in order to support `class_aliases`, etc.
rcn	Remove concatenations with `""`.
desug	Desugar: Canonicalize simple constructs.
sma	Split multiple arguments for globals, attributes and static declarations, into multiple declarations with single arguments.
sui	Split **`unset`**`()` and **`isset`**`()` into multiple calls with one argument each.
ecs	Split **`echo`**`()` into multiple calls with one argument each.
elcf	Early Lower Control-Flow: simplify **`for`**, **`while`**, **`do`**-**`while`** and **`switch`** statements.
lef	Lower Expression Flow: lower `||`, `&&` and `?:` expressions.
lish	List Shredder: convert list assignments into a set of array assignments.
shred	Shredder: turn the AST into three-address-code, extracting complex expressions, and inserting temporary variables.
pps	Convert postfix unary operations into prefix operations.
swbin	Replace `>=` and `>` binary operations with `<` and `<=`, switching their operands.
rse	Remove expressions whose result is not stored, and which are side-effect free.

Table 4.2: *Passes to lower the AST into HIR*

prc	Propagate Copies: Perform simple copy propagation to remove some copies of temporary variables introduced as a result of lowering.
dtc	Dead Temp Cleanup: remove assignments to temporary variables which are not used.
ldd	Lower Dynamic Definitions: Lower dynamic class, interface and method definitions using aliases, moving and renaming the definition.
lmi	Lower Method Invocations: Lower parameters after a run-time reference check, using `param_is_ref`.
lcf	Lower Control Flow: Replace all control-flow statements with `gotos`.

Table 4.3: *Passes to lower the HIR into MIR*

obfuscate	Optionally print program with obfuscated control-flow.
lfc	Move statements from global scope into __MAIN__ method.
clar	Clarify: Make implicit statements explicit.
pst	Prune Symbol Table: Note whether a symbol table is required in generated code.

Table 4.4: *MIR passes*

cfg	Build Control-Flow Graph.
build-ssa	Build SSA form.
ifsimple	If-simplification: Remove negation by switching branch targets.
dce	Aggressive Dead-code Elimination.
drop-ssa	Drop SSA form.
rlb	Remove Loop Booleans: Clone and specialize loop entries to remove redundant boolean variables.
inlining	Method inlining: Inline simple methods at monomorphic call-sites.

Table 4.5: *Optimization passes*

4.5 Embed System

The most important design decision in phc is to add a facility to call into the PHP system. Section 5 describes the reasons for which phc uses the PHP system. In particular, the link between the PHP system and phc is described in Section 5.4.2. In this section, we explore the uses of the PHP system throughout phc.

4.5.1 Language Flags

The PHP language can be configured using a number of language flags. These flags can be provided on the command-line, in system configuration files, or at run-time. The PHP system automatically reads from system configuration files when phc starts up, and adapts according to changes at run-time. As the behaviour of the PHP system—and the results of its functions—often depend on these '*.ini*' settings, phc also allows these to be set on the command-line. These flags are fed into the PHP system, and subsequent calls into the PHP system reflect their presence.

cgann	Make annotations for code generation.
generate-c	Generate C code from the MIR.
compile-c	Compile C code into an executable.

Table 4.6: *Code generation passes*

Possibly the most important flag is the `include_path` flag. This provides a list of directories to search when including files. Not all of these flags can be used, however. Some settings—such as `zend.ze1_compatibility_mode`, which changes PHP's memory behaviour to be compatible with that of PHP 4—affect language semantics, and are unsupported.

4.5.2 Evaluating Expressions

The most important use of the PHP system at compile-time is to determine the values of PHP expressions. During parsing, the values of all literals are parsed using the PHP system, as described in Section 5.2.2. During optimization, the PHP system is used to fold arithmetic, logical and string operations, to evaluate casts between known scalar values. In particular, it simplifies the task determining whether a scalar has a true value, or whether two scalars are equal using PHP's weak equality rules.

4.5.3 Querying PHP Implementation Information

Finally, the PHP system allows `phc` to query values which would be resolved at run-time, a form of partial evaluation. During optimization, we can access the values of constants that are built into PHP. To resolve whether a function's parameters are passed by reference or by copy, we query the PHP system. During code generation, we can use a string's hash value to optimize the generated code, as described in Section 5.4.5.

4.6 Optimization

This section provides an overview of all the optimization in `phc`, focussing on the various parts of `phc` which benefit from simple or advanced optimization.

4.6.1 Early Optimization

Two passes provide early optimizations. The *remove-concat-null* pass (named '*rcn*' in Figure 4.2) removes concatenations with empty strings. Superfluous concatenations

are introduced by the phc parser in parsing string interpolation.[3] In theory, these concatenations might be considered identity operations, as

```
"" . $x
```

should be equivalent to `$x`. However, concatenation is also used to get the string value of `$x`, which is not correct to remove. As a result, the *remove-concat-null* pass aims only to remove those assignments which are truly identity operations. It is performed early in the compilation pipeline so that the parsing annotations are still present, and the annotations must be consulted to ensure that only correct operations are removed.

The *const-fold* pass performs constant folding, by consulting the PHP system for the result of simple operations between literals, and replacing the operation with a new literal, in the AST.

Not all constants can be folded in this manner. A division-by-zero leads to a run-time warning, for example, so we detect warnings and leave the original operation alone if they occur. The *const-fold* pass does not attempt to fold identity operations, such as `$x * 1`, for the same reason as the *remove-concat-null* pass.

4.6.2 Cleanup Optimizations

A number of phc's optimization passes are intended to tidy the results of other passes, which may not be tuned for optimal code. This is relatively standard in optimizing compilers.

- During lowering from the AST to the HIR, a number of conditions can lead to extra copies between temporary variables being added. These extra copies can lead to severe slowdown because the PHP system uses '*copy-on-write*' during assignments. This is described in more detail in Section 5.5.2, with an example.

  To deal with this problem, we add simple copy-propagation and dead-code elimination passes to the HIR. As we cannot detect aliasing in the HIR, this analysis is very simple. Only temporary variables which are defined and used exactly once are considered for copy propagation. Similarly, dead code elimination is only performed on temporary variables that are only defined once by copy statements, and which are never used.

[3] String interpolation is a string formatting technique where variables (and sometime more complex expressions) can be embedded within a literal string, for example `"DEBUG: $level: $msg"`.

```
$T = !$T2;
if ($T)
    echo "path A";
else
    echo "path B";
```

(a)

```
if ($T)
    echo "path B";
else
    echo "path A";
```

(b)

Listing 4.6: *A demonstration of the* if-simplification pass

- The *ifsimple* pass performs *if-statement simplification*, by removing the combination of negation and branches. In cases where a branch condition is only set once to a negation of another variable, such as in Listing 4.6, we replace the original variable with the negated one, and swap the branch targets. This allows lowering passes to use statements such as `if ($T2)` without worrying about the consequences.

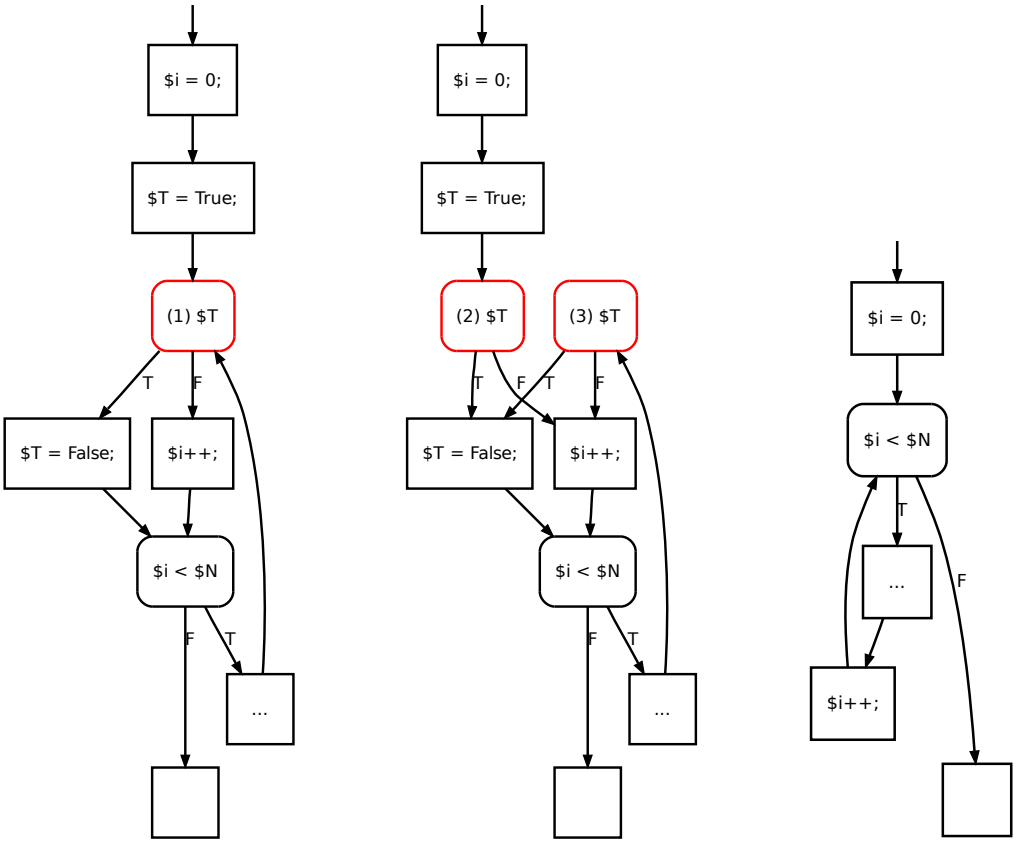
- During the conversion to a single loop type in the HIR, `phc` introduces a number of conditional statements into the program, as shown in '*Loop design*' in Section 4.2.4. However, these conditional statements could have been omitted, if `phc` proceeded straight to an MIR in place of using the HIR; the MIR would have used `goto`s instead, and the conditional statements would not have been necessary.

  The *remove-loop-booleans* pass (called '*rlb*' in Figure 4.5) helps solve the problem, as shown in Figure 4.2. This figure depicts the control-flow graph for the snippet in Listing 4.5.

  The pass identifies the branch which uses the boolean value (marked (1) in Figure 4.2a). This branch has two predecessors, one from each of the head and tail of the loop. The optimization clones the block, so that there is a separate block for each predecessor. These cloned blocks are marked (2) and (3) in Figure 4.2b. It is known how each of the cloned branches will be resolved: (2) must always be true, and (3) must always be false. Therefore both the branches and the definitions of the boolean variables may be removed by unreachable- and dead-code elimination, resulting the Figure 4.2c. The end result is the same as if the loops were converted directly into `goto` statements in the MIR.

  This is similar to unrolling a loop's first iteration, in order to optimize values and types which are invariant after the first iteration, as discussed in Section 6.6.

- The optimizer performs a powerful dead-code elimination, discussed in Section 6.5.5. It also performs simple control-flow optimizations [Torczon and



(a) *Before optimization.*

(b) *The branch is cloned, with one predecessor for each branch.*

(c) *After the dead- and unreachable code is eliminated.*

Figure 4.2: *Removal of conditional statements in the* remove loop booleans *pass*

Cooper, 2007, Section 10.3.1] which remove empty or unreachable basic blocks, and redundant edges.

4.6.3 Whole-program Analysis

The phc compiler features an advanced optimization framework. At its core is a combined alias-, type- and constant-analysis, described in detail in Section 6.5. This section describes the design and structure of the optimization framework.

Overview

After the program is lowered into MIR, each function and class in the program are stored. They are processed lazily, beginning with the `__MAIN__` function. A method is converted into a CFG, then processed using a worklist algorithm, beginning with the entry block. The worklist algorithm is based on that of Pioli et al. [1999], which is described in Section 6.5.1.

Upon reaching a function or method call, the potential callees are fetched, and the analysis continues in the callee. Upon exhausting the statements in a CFG's worklist, the analysis continues in the caller, or ends in the case of the `__MAIN__` method.

The main unit of analysis is the *name*, meaning some memory location in the program, such as a variable, an object's field or an array offset. A more thorough definition of a *name* can be found in Section 6.5.

Client Analyses

The framework is structured as a driver with a set of client analyses. Each statement is analysed by the driver, and is decomposed into a number of different operations. Each of these operations is passed to the client analyses. The client analyses are:

- **alias analysis**: builds a points-to graph of the program state,
- **value propagation**: stores a lattice (see Section 6.5.3) of literals and a set of possible types for each *name* in the program,
- **use-def analysis**: stores a set of names that are used and defined in each statement. This is used to build an SSA form later. This cannot be built syntactically, as described in Section 6.5.5.
- **constant propagation**: similar to value propagation—and uses the same lattice—but for PHP constants (which are defined at most once per program) instead of names.

It is possible to use other client analyses using the same interface. Suitable analyses for future work include value-range propagation [Patterson, 1995] and string-range propagation [Wassermann and Su, 2007]. The API used by the client sub-analyses is provided in Appendix B.

Assignments

For each statement in the MIR analysed, the effects of the statement on the program are modelled. Consider for example a simple assignment statement `$x = $y;`. The semantics of this simple assignment are to copy the value of `$y` into `$x`. `$x` may reference other variables; if so, `$y` is also copied into those referenced names. The algorithm to model a simple assignment is as follows:

```
// Get aliases of $x
lhss = get_aliases ("x")

// If there can be no aliases, remove the old value.
if size (lhss) == 1
    && parent(lhss[0]) is not abstract:
    kill_value (lhss[0])

foreach lhs in lhss:
    copy_value (lhs, "y");
```

All other assignments can be modelled with slight variations, such as reading from an array index or expression instead of from a simple variable.

Reference assignments (`$x =&$y;`) are slightly different, however. If `$x` references other variables, those references are broken, and a new reference is created with `$y`.

Optimization Uses

Many small optimizations are performed using the results of the whole-program analysis. Most of these optimizations use the results of literal analysis[4] in order to make statements redundant. Appendix C lists the transformations that `phc` performs as a result of its optimization. It also lists optimizations which could be implemented, and optimizations which could be performed with small extensions to the analysis.

[4] In PHP, *constants* have a particular meaning, defined in Section 6.2.7. As a result, we use the term '*literal*' where the term *constant* might often be used. This term is used in previous work on PHP [Jovanovic et al., 2006].

4.6.4 SSA Form

The results of the def-use analysis are used to build a *Static Single-Assignment* (SSA) form. The phc compiler uses an SSA form based on *Hashed SSA* (HSSA) [Chow et al., 1996]. We discuss the challenge of building an SSA form, and using it for dead-code elimination in Section 6.5.5.

HSSA has quite a number of features which we do not use. Although our SSA form more closely resembles HSSA form than any other kind of SSA form, in practice we use only a small number of its features. The main idea which we take from HSSA is that all memory locations are given names. We also use the μ and χ[5] operations. We do not use virtual variables, zero versioning, or global value numbering (the '*hash*' in '*Hashed SSA*'). However, in order to scale our SSA form in the future, it may become useful to add these features.

Principally, the difference between our SSA form and traditional SSA is that all names in the program are included in the SSA representation. In addition, for each statement in the program, we maintain a set of names which must be defined, which might be defined, and which are used (either possibly or definitely). Names which may be defined are the same concept as χ variables in HSSA, but there may also be more than one name which *must* be defined.

We also add ϕ-functions for names which are only used in χ operations, or which must be defined but are not syntactically present in the statement. As these ϕ-functions are not for real variables, we cannot add copies to represent them when moving out of SSA form, and must instead drop all SSA indices. While this is a valid means of moving out of SSA form, it slightly limits the optimizations that can be performed. For example, copy-propagation cannot move copies past the boundary of a variable's live range.

Constructing SSA Form on Demand

In phc, we perform propagation analyses, such as conditional constant propagation and type inference, before creation of SSA form. This makes the results of those analyses available at the same time as alias analysis, and provides important feedback for the optimization framework. However, this ordering of analyses means that we cannot benefit from some of the advantages of SSA, such as its sparse framework for

[5] '*mu*' and '*chi*', respectively.

propagation analyses.

It would be desirable to avoid this *phase ordering* problem. In order to do so, it is necessary to be able to build SSA form as a client analysis which can be run alongside other analyses. In order to achieve this the following SSA construction algorithm, which uses symbolic execution, should work as a first step. It is presented in this section.[6]

The symbolic execution algorithm, of which the new SSA construction algorithm is a client, is part of the SCCP algorithm [Wegman and Zadeck, 1991]. As a first step, I generalize the SCCP algorithm:

1. Use two worklists, one for CFG edges (from one basic block to another), and one for SSA edges (from a definition to a use).
2. Starting at the entry node of the CFG worklist, analyse each statement s of a basic block bb, then:
   - For each variable definition d in s, add all uses of d to the SSA worklist,
   - Add each successor $succ$ of bb to the CFG worklist, only if $succ$ is executable according to the analysis so far.
3. Once the CFG worklist is exhausted, analyse each statement s which is the target of an edge in the SSA worklist, then:
   - As above, for each variable definition d in s, add all uses of d to the SSA worklist.

This generalized form of the SCCP algorithm can be used for many client analyses. Our novel SSA construction algorithm extends it to allow SSA construction via symbolic analysis, in the following way:

Assumptions

- Dominance information (the immediate dominator and dominance frontiers of each basic block) is available.
- All variables in our program begin out of SSA form, and ϕ nodes are not initially present anywhere in the program.

[6] Note that phc does not actually use this algorithm yet, as it has not been integrated with other analyses.

- When ϕ nodes are added, they are placed as the first statements in the basic block.

Algorithm

Following the SCCP algorithm, for each statement *s* in a basic block *bb*:

1. For each use *u* in *s*:
    - Fetch an SSA version for *u* by searching upwards through the dominance tree of *bb* for definitions of *u*:
        - If no definitions are found, the zero version if chosen,
        - If *s* is a ϕ node, we begin the search in the predecessor of *bb* associated with *u*, rather than *bb* itself,
    - If *u* already has a version, we must still recalculate it, because a definition[7] may have been inserted in between *s* and the previous definition of *u*.
2. For each def *d* in *s*:
    - Add an unversioned ϕ node *p*, named after *d*, to each dominance frontier of *b*, and add *p* to the SSA worklist,
    - Give a new version to *d*.
    - Add the following uses of *d* to the SSA worklist:
        - The unversioned *d*, from before SSA construction, as they might not all have been reached,
        - The version of *d* which holds just before the new definition, as this *s* may be a new statement inserted on an existing path between some definition of *d* and its use.

In order to be used for `phc`, this algorithm needs to run simultaneously with alias analysis, constant-propagation, and other client analyses of the analysis framework. This is future work, however.

[7] Due to the addition of a ϕ node, or a χ node in HSSA form.

4.7 Conclusion

This chapter describes the design of the phc compiler, including the careful design of the phc intermediate representations. The design of the whole-program optimization framework and SSA analysis is presented, as well as how to extend it to build an SSA form on demand, using a symbolic execution framework. We provide a description of important optimizations which mitigate the weaknesses imposed on use during the design of the phc intermediate representations, It also discusses how to compile a language designed for interpretation, and the mechanism with which to call into the PHP system. The next chapter goes in to more detail on this topic, focussing on the PHP system, and how it is combined into both phc, and the code generated by phc.

A Practical Solution for Scripting Language Compilers

Although scripting languages are becoming increasingly popular, even mature scripting language implementations remain interpreted. Several compilers and reimplementations have been attempted, generally focusing on performance.

Based on our survey of these reimplementations, we determine that there are three important features of scripting languages that are difficult to compile or reimplement. As scripting languages are defined primarily through the semantics of their original implementations, they often change semantics between releases. They provide C APIs, used both for foreign-function interfaces and to write third-party extensions. These APIs typically have tight integration with the original implementation, and are used to provide large standard libraries, which are difficult to re-use, and costly to reimplement. Finally, they support run-time code generation. These features make the important goal of correctness difficult to achieve for compilers and reimplementations.

We present a technique to support these features in an ahead-of-time compiler for PHP. Our technique uses the original PHP implementation through the provided C API, both in our compiler, and in our generated code. We support all of these important scripting language features, particularly focusing on the correctness of compiled programs. Additionally, our approach allows us to automatically support limited *future* language changes. We present a discussion and performance evaluation of this technique.

5.1 Motivation

Although scripting languages[1] are becoming increasingly popular, most scripting language implementations remain interpreted. Typically, these implementations are slow, between one and two orders of magnitude slower than C. There are a number

[1] The techniques in this chapter apply to PHP, Perl, Python, Ruby and Lua. However, they do not apply to Javascript because it does not share many of the attributes we discuss in this chapter. Notably, it is standardized, and many distinct implementations exist, none of which are canonical.

of reasons for this. Scripting languages have grown up around interpreters, and were generally used to glue together performance sensitive tasks. Hence, the performance of the language itself was traditionally not important. As they have increased in prominence, larger applications are being developed entirely in scripting languages, and performance is increasingly important.

The major strategy for retrofitting performance into an application written in a scripting language is to identify performance hot-spots, and rewrite them in C using a provided C API. Modern scripting languages are equipped with C APIs which can interface with the interpreter, in fact, in many cases the interpreters themselves are written using these APIs. Though this is not a bad strategy—it is a very strong alternative to rewriting the entire application in a lower level language—a stronger strategy may be to compile the entire application. Having a compiler automatically increase the speed of an application is an important performance tool, one that contributes to the current dominance of C, C++ and Java.

However, it is not straightforward to write a scripting language compiler. The most important attribute of a compiler—more important than speed—is correctness, and this is difficult to achieve for a scripting language. Scripting languages do not have any standards or specifications.[2] Rather, they are defined by the behaviour of their initial implementation, which we refer to as their '*canonical implementation*'.[3] The correctness of a later implementation is determined by its semantic equivalence with this canonical implementation. It is also important to be compatible with large standard libraries, written in C. Both the language and the libraries often change between releases, leading to not one, but multiple implementations with which compatibility must be achieved.

In addition, there exist many third-party extensions and libraries in wide use, written using the language's built-in C API. These require a compiler to support this API in its generated code, because reimplementing the library may not be practical, especially if it involves proprietary code.

A final challenge is that of run-time code generation. Scripting languages typically support an `eval` construct, which executes source code at run-time. Even when `eval` is not used, the semantics of some language features require some computation to be deferred until run-time. A compiler must therefore provide a run-time component,

[2] This is becoming less true for Python and Lua, which now provide reference manuals.

[3] A canonical implementation differs subtly from a reference implementation, in that a reference implementation provides an implementation of a specification, while a canonical implementation provides the specification.

with which to execute the code generated at run-time.

In phc we are able to deal with the undefined and changing semantics of PHP by integrating the PHP code. At compile-time, we use the PHP system as a language oracle, giving us the ability to automatically adapt to changes in the language, and allowing us avoid the long process of documenting and copying the behaviour of myriad different versions of the language. We also generate C code which interfaces with the PHP system via its C API. This allows our compiled code to interact with built-in functions and libraries, saving not only the effort of reimplementation of large standard libraries, but also allowing us to interface with both future and proprietary libraries and extensions. Finally, we reuse the existing PHP system to handle run-time code generation, which means we are not required to provide a run-time version of our compiler, which can be a difficult and error-prone process.

As many of the problems we discuss in this chapter occur with any reimplementation, whether it is a compiler, interpreter or JIT compiler, we shall generally just use the term '*compiler*' to refer to any scripting language reimplementation. We believe it is obvious when our discussion only applies to a compiler, as opposed to a reimplementation which is not a compiler.

In Section 5.2.1, we provide a short motivating example, illustrating these three important difficulties: the lack of a defined semantics, emulating C APIs, and supporting run-time code generation. In Section 5.3, we examine a number of previous scripting language compilers, focusing on important compromises made by the compiler authors which prevent them from correctly replicating the scripting languages they compile. Our approach is discussed in Section 5.4, explaining how each important scripting language feature is correctly handled by re-using the canonical implementation. Section 5.5 discusses PHP's memory model. Section 5.6 discusses the complementary approach of using a JIT compiler. An experimental evaluation of our technique is provided in Section 5.7, including performance results, and supporting evidence that a large number of programs suffer from the problems we solve.

5.2 Challenges to Compilation

There are three major challenges to scripting languages compilers: the lack of a defined semantics, emulating C APIs, and supporting run-time code generation. Each presents a significant challenge, and great care is required both in the design and

implementation of scripting language compilers as a result. We begin by presenting a motivating example, before describing the three challenges in depth.

5.2.1 Motivating Example

Listing 5.1 contains a short program segment demonstrating a number of features which are difficult to compile. The program segment itself is straightforward, loading an encryption library and iterating through files, performing some computation and some encryption on each. The style uses a number of features idiomatic to scripting languages. Though we wrote this program segment as an example, each important feature was derived from actual code we saw in the wild.

Lines 3-6 dynamically load an encryption library; the exact library is decided by the `$engine` variable, which may be provided at run-time. Line 9 creates an array of hexadecimal values, to be used later in the encryption process. Lines 12-16 read files from disk. The files contain data serialized by the **`var_export`** function, which converts a data structure into PHP code which when executed will create a copy of the data structure. The serialized data is read on line 16, and is deserialized when line 17 is executed. Lines 20-28 represent some data manipulation, with line 20 performing a hashtable lookup. The data is encrypted on line 31, before being re-serialized and written to disk in lines 34 and 35 respectively. Line 37 selects the next file by incrementing the string in `$filename`.

5.2.2 Undefined Language Semantics

A major problem for reimplementations of scripting languages is the languages' undefined semantics. Jones [2008] describes a number of forms of language specification. Scripting languages typically follow the method of a '*production use implementation*' in his taxonomy. In the case of PHP, Jones says:

> "The PHP group claim that they have the final say in the specification of PHP. This group's specification is an implementation, and there is no prose specification or agreed validation suite. There are alternate implementations [...] that claim to be compatible (they don't say what this means) with some version of PHP."

```php
define(DEBUG, "0");

# Create instance of cipher engine
include 'Cipher/' . $engine . '.php';
$class = 'Cipher_' . $engine;
$cipher = new $class();

# Load s_box
$s_box = array(0x30fb40d4, ..., 0x9fa0ff0b);

# Load files
$filename = "data_1000";
for($i = 0; $i < 20; $i++)
{
    if(DEBUG) echo "read serialized data";
    $serial = file_get_contents($filename);
    $deserial = eval("return $serial;");

    # Add size suffix
    $size =& $deserial["SIZE"];
    if ($size > 1024 * 1024 * 1024)
        $size .= "GB";
    elseif ($size > 1024 * 1024)
        $size .= "MB";
    elseif ($size > 1024)
        $size .= "KB";
    else
        $size .= "B";

    # Encrypt
    $out = $cipher->encrypt($deserial, $s_box);

    if(DEBUG) echo "reserialize data";
    $serial = var_export($out, 1);
    file_put_contents($filename, $serialized);

    $filename++;
}
```

Listing 5.1: *PHP code demonstrating dynamic, changing or unspecified language features*

As a result of this lack of abstract semantics, compilers must instead adhere to the concrete semantics of the canonical implementation for correctness. However, different releases of the canonical implementation may have different concrete semantics. In fact, for PHP, changes to the language definition occur as frequently as a new release of the PHP system. In theory, the language would only change due to new features. However, new features frequently build upon older features, occasionally changing the original semantics. Older features are also modified with bug fixes. Naturally,

changes to a feature may also introduce new bugs, and there exists no validation suite to prevent these bugs from being considered features. In a number of cases we have observed, a '*bug*' has been documented in the language manual, and referred to as a feature, until a later release when the bug was fixed. As a result of these changes, even the same feature in different versions of the language may have different semantics.

While in a standardized language, such as C or C++, the semantics of each feature is clearly defined,[4] in a scripting language, the task of determining the semantics can be arduous and time consuming. Even with the source code of the canonical implementation available, it is generally impossible to guarantee that the semantics are copied exactly.

Literal Parsing

A simple example of a change to the language is a bug fix in PHP version 5.2.3, which changed the value of some integer literals. In previous versions of PHP, integers above `LONG_MAX`[5] were converted to floating-point values—unless they were written in hexadecimal notation (e.g. `0x30fb40d4`). In this case, as in our example on line 9 of Listing 5.1, they were to be truncated to the value of `LONG_MAX`. Since version 5.2.3, however, these hexadecimal integers are converted normally to floating-point values.

Libraries

One of the major attractions of scripting languages is that they come '*batteries included*', meaning they support a large standard library. However, unlike the C++ or Java standard libraries, a scripting language's standard library is typically written in C, using the C API. Compilers which do not emulate the C API must instead reimplement the libraries. As the libraries are not specified, they are liable to change, and new libraries are constantly being added.

[4] Standardized languages also consider some semantics 'undefined', meaning an implementation can do anything in this case. No scripting language features are undefined, as they all do *something* in the canonical implementation.

[5] Constant from the C standard library representing the maximum signed integer representable in a machine word.

Built-in Operators

The lack of abstract semantics also means that it is difficult to know the exact behaviour of some language constructs, especially due to PHP's implicit type conversions (see Section 6.2.8). Addition, for example, is more general in PHP than in C. Its behaviour depends on the run-time type of the operands, and overflows integers into floats.[6] There is a significant amount of work in determining the full set of semantics for each permutation of operator and built-in type. What, for example, is the sum of the string `"hello"` and the boolean value **`true`**?[7] As another example, the two statements `$a = $a + 1;` and `$a++;` are not equivalent. The latter will '*increment*' strings, increasing the ASCII value of the final character, another unlikely language feature, as shown in Listing 5.1 on line 37.

Truth is also complicated in PHP, due to its implicit type conversions. Conditional statements implicitly convert values to booleans, and the conversions are not always intuitive. Example of false values are `"0"`, `""`, `0`, **`false`** and `0.0`. Examples of true values are `"1"`, `1`, **`true`**, `"0x0"` and `"0.0"`.

Language Flags

In PHP, the semantics of the language can be tailored through use of the `php.ini` file. Certain flags can be set or unset, which affect the behaviour of the language.

The `include_path` flag affects separate compilation, and alters where files can be searched for to include them at compile time. The `call_time_pass_by_ref` flag decides whether a caller is permitted to pass its actual parameter to a function by reference, potentially overriding the function's default of passing by-copy.

5.2.3 C API

A scripting language's C API provides its foreign-function interface. Typically, it is used for embedding the language into an application, creating extensions for the language, and writing libraries. A discussion of the merits of various scripting languages' C APIs is available [Muhammad and Ierusalimschy, 2007].

[6] Feeley [2007] discusses a similar problem in Scheme, in that several Scheme compilers incorrectly prevent integers from overflowing into *Bignums* for performance reasons.
[7] An integer `1`, it seems.

Typically, the C API is the only part of the language with stable behaviour. Though features are added over time, the C API is in such heavy use that regressions and bugs are noticed quickly. We have seen that even when changes to the language and its libraries are frequent, changes to the behaviour of the C API are not.

5.2.4 Run-time Code Generation

A number of PHP's dynamic features allow source code, constructed at run-time, to be executed at run-time. Frequently these features are used as quick hacks, and they are also a common vector for security flaws. However, there are a sufficient number of legitimate uses of these features that a compiler must support them.

Eval Statements

As demonstrated in Listing 5.1, the `eval` statement executes arbitrary fragments of PHP code at run-time. It is passed a string of code, which it parses and executes in the current scope, potentially defining functions or classes, calling functions whose names are passed by the user, or writing to user-named variables.

Include Statements

The PHP `include` statement is used to import code into a given script from another source file. Although similar in theory to the `eval` statement, this feature is generally used by programmers to logically separate code into different source files, in a similar fashion to C's `#include` directive, or Java's `import` declaration. However, unlike those static approaches, an `include` statement is executed at run-time, and the included code is only then parsed and executed.

Dynamic `include` statements are commonly used in PHP to provide a plugin facility, or to implement localization. In several cases we have observed, a program's runtime-configuration file (e.g. Unix `.rc files`) is simply a Python file which is run at program startup. In Section 5.7.5, we provide statistics about usage of dynamic and static includes (as well as `eval` statements) from a large number of publicly available PHP programs.

Variable-variables

PHP variables are simply a map of strings to values. Variable-variables provide a means to access a variable whose name is known at run-time—for example, one can assign to the variable `$x` using a variable containing the string value `"x"`. Access to these variables may be required by `eval` or `include` statements, and so this feature may take advantage of the infrastructure used by these functions. Variable functions are also accessible in this way, and Listing 5.1 shows a class initialized dynamically in the same manner.

5.3 Related Work

Having discussed scripting language features which can be challenging to compile, we examine previous scripting language compilers, discussing how they handled these features in their implementations. We believe that many of their solutions are sub-optimal, either requiring great engineering or sacrifices which limit the potential speed improvement of their approach.

5.3.1 Undefined Language Semantics

The most difficult and rarely addressed issue is ensuring that a program is executed correctly by a reimplementation of a scripting language. In particular, it is rarely mentioned that different versions of a scripting language can have different semantics, especially in standard libraries.

Very few scripting language compilers provide any compatibility guarantees for their language. Instead, we very often see laundry lists of features which do not work, and libraries which are not supported. A number of implementations we surveyed chose to rewrite the standard libraries. UCPy [Aycock et al., 2003], a reverse-engineered Python compiler, reports many of the same difficulties that motivated us: a large set of standard libraries, a language in constant flux, and a manual whose contents surprise its own authors. They chose to rewrite the standard library, even though it was 71,000 lines of code long, risking potential semantic differences with the official distribution.

Both Roadsend [Roadsend] and Quercus [Quercus]—PHP compilers, referred to by

Jones' quote in Section 5.2.2—reimplement a very small portion of the PHP standard libraries. In Shed Skin [Dufour, 2006, Section 4.3.3], a Python-to-C++ compiler, the authors were unable to analyse or reuse Python's comprehensive standard library. Instead, library functions they wanted to support were both reimplemented in C++ and separately modelled in Python.

Jython [Jython] and JRuby [JRuby] are reimplementations of Python and Ruby, respectively, on the JVM. They reimplement their respective standard libraries in their respective host languages, and do not reuse the canonical implementation. A much better approach is employed by Phalanger [Benda et al., 2006, Section 3], a PHP compiler targeting the *.NET* run-time. It uses a special manager to emulate the PHP system, through which they access the standard libraries via the C API. They report that they are compatible with the entire set of PHP extensions and standard libraries. However, Phalanger does not use the PHP system's functions for its built-in operators, instead rewriting them in its host language, *C#*. Many of PHP's most difficult features to compile involve its built-in operators, and we believe that reimplementing them is costly and error-prone.

In terms of language features, none of the compilers discussed have a strategy for automatically adapting to new language semantics. Instead, each provides a list of features with which they are compatible, and the degree to which they are compatible. None mentioned the fact that language features change, or that standard libraries change, and we cannot find any discussion of policies to deal with these changes.

A few, however, mention specific examples where they were unable to be compatible with the canonical implementation of their language. Johnson and Slattery [2006] attempted to reimplement PHP from public specifications, using an existing virtual-machine. They reported problems caused by PHP's call-by-reference semantics. In their implementation, callee functions are responsible for copying passed arguments, but no means was available to inform the callee that an argument to the called function was passed-by-reference.[8] Shed Skin [Dufour, 2006] deliberately chose to use restricted language semantics, in that it only compiles a statically typed subset of Python.

However, two approaches stand out as having taken approaches which can guarantee a strong degree of compatibility. 211 [Aycock, 1998] converts Python virtual machine code to C. It works by pasting together code from the Python interpreter, which corresponds to the bytecodes for a program's hot-spots. 211 is a compiler which is very resilient to changes in the language, as its approach is not invalidated by the

[8] In PHP, call-by-reference parameters can be declared at function-definition time or at call-time.

addition of new opcodes. It's approach is more likely to be correct than any other approach we mention, including our own, though it comes at a cost, which we discuss is Section 5.7.2.

Python2C [Salib, 2004, Section 1.3.1] has a similar approach to `phc`, and, like both `phc` and 211, provides great compatibility. Unfortunately, it comes with a similar cost to 211, as detailed in Section 5.7.2.

Pyrex [Ewing] is a domain-specific language for creating Python extensions. It extends a subset of Python with C types and operations, allowing mixed semantics within a function. It is then compiled, in a similar fashion to our approach. Though they omit much of the language, it is easy to see that by following this approach, they have to ability to have a very high degree of compatibility with Python, even as the language changes.

5.3.2 C API

Very few compilers attempt to emulate the C API. However, Johnson and Slattery [2006] provide a case study, in which they determine that it is not possible in their implementation, claiming that the integration between the PHP system and the extensions was too tight. We have also observed this, as the C API is very closely modelled on the PHP system's implementation. Phalanger [Benda et al., 2006] does not emulate the C API, but it does provide a bridge allowing programs to call into extensions and libraries. Instead of a C API, it provides a foreign-function interface through the .Net run-time. Jython and JRuby provide a foreign-function interface through the JVM, in a similar fashion.

5.3.3 Run-time Code Generation

A number of compilers [Benda et al., 2006, Johnson and Slattery, 2006, JRuby, Jython, Roadsend] support run-time code generation using a run-time version of their compiler. Some [Dufour, 2006, Quercus] choose not to support it at all. Quercus in particular claims not to support it for security reasons, as run-time code generation can lead to code-injection security vulnerabilities. We show in Section 5.7.5 that this results in a large number of PHP programs which could not be run using the Quercus compiler.

While providing a run-time portion of the compiler is sensible for a JIT compiler, which has already been designed as a run-time system, most of these implementations are not JIT compilers. However, providing this run-time portion requires that the implementation is suitable for run-time use; it must have a small footprint, it cannot leak memory, it must be checked for security issues, and it must generate code which interfaces with the code which has already been generated. These requirements are not trivial, and we believe the approach we outline in Section 5.4 affords the same benefits, at much lower engineering cost. We discuss using a JIT compiler in more detail in Section 5.6.

5.3.4 Other Approaches

Walker and Griswold's [1992] optimizing compiler for Icon uses the same system for its compiled code as its interpreter used. In addition, as they were in control of both the compiler and the run-time system, they modified the system to generate data to help the compiler make decisions at compile-time. Typically, scripting language implementations do not provide a compiler, and compilers are instead created by separate groups. As a result, it is generally not possible to get this tight integration, though it would be the ideal approach.

5.4 Our Approach

Nearly all of these approaches have been deficient in some manner. Most were not resilient to changes in their target language, and instead reimplemented the standard libraries [Aycock et al., 2003, Dufour, 2006, Johnson and Slattery, 2006, JRuby, Jython, Quercus, Roadsend]. Those which handled this elegantly still failed to provide the C API [Benda et al., 2006], and those which achieved a high degree of compatibility [Aycock, 1998, Ewing, Salib, 2004] failed to provide a means to achieving good performance.

In phc, our ahead-of-time compiler for PHP, we are able to correct all of these problems by integrating the PHP system into both our compiler and compiled code. At compile-time, we use the PHP system as a language oracle, allowing us to automatically adapt to changes in the language, and saving us the long process of documenting and copying the behaviour of many different versions of the language. Our generated C code interfaces with the PHP system at run-time, via its C API. This allows our compiled

code to interact with built-in functions and libraries and to re-use the existing PHP system to handle run-time code generation.

5.4.1 Undefined Language Semantics

Language Semantics

One option for handling PHP's volatile semantics is to keep track of changes in the PHP system, with separate functionality for each feature and version. However, our link to the PHP system allows us to resiliently handle both past and future changes.

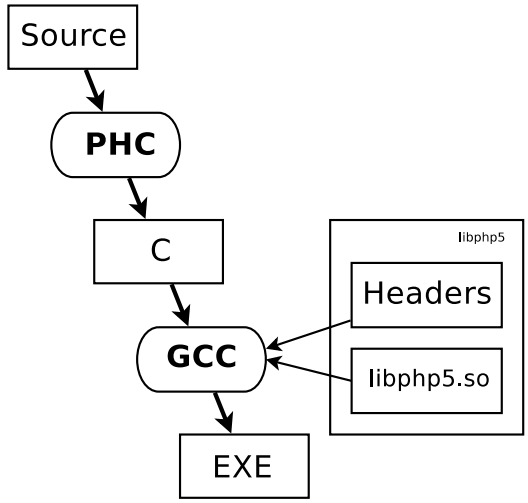
For built-in operators, we add calls in our generated code to the built-in PHP function for handling the relevant operator. As well as automatically supporting changes to the semantics of the operators, this also helps us avoid the difficulty of documenting the many permutations of types, values and operators, including unusual edge cases.

We solve the problem of changing literal definitions by parsing the literals with the PHP system's interpreter, and extracting the value using the C API. If the behaviour of this parsing changes in newer versions, the new PHP system's interpreter will still parse it correctly, and so we can automatically adapt to some language changes which have not yet been made.

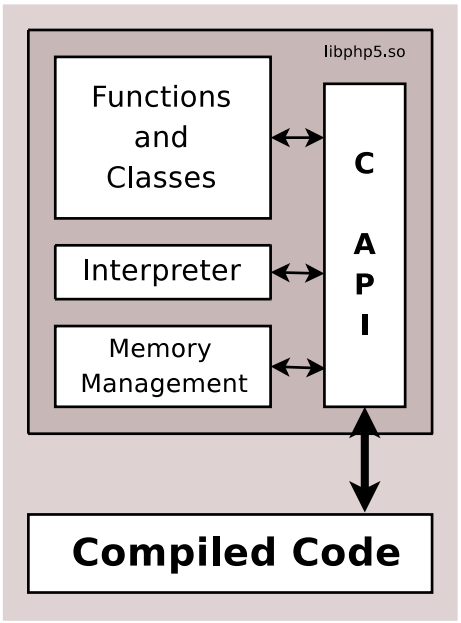
We handle language flags by simply querying them via the C API. With this, we can handle the case where the flag is set at configure-time, build-time, or via the `php.ini` file. No surveyed compiler handles these scenarios.

Libraries and Extensions

One of the largest and most persistent problems in creating a scripting language reimplementation is that of providing access to standard libraries and extensions. We do not reimplement any libraries or extensions, instead re-using the PHP system's libraries via the C API. This allows us to support proprietary extensions, for which no source code is available, which is not possible without supporting the C API. It also allows support for libraries which have yet to be written, and changing definitions of libraries between versions.



(a) *Compilation pipeline with* phc *and the PHP C API*



(b) *Interface between the* phc *compiled code and the PHP C API*

Figure 5.1: *Interaction of* phc *and the PHP C API*

5.4.2 C API

Naturally, we support the entire C API, as our generated code is a client of it. This goes two ways, as extensions can call into our compiled code in the same manner as the code calls into extensions.

Integrating the PHP system into the compiler is not complicated, as most scripting languages are designed for embedding into other applications [Muhammad and Ierusalimschy, 2007]. Lua in particular is designed expressly for this purpose [Ierusalimschy et al., 2005]. In the case of PHP, it is a simple process [Golemon, 2006] of including two lines of C code to initialize and shut down the PHP system. We then compile our compiler using the PHP *embed* headers, and link our compiler against the *embed* version of `libphp5.so`, the shared library containing the PHP system. This is shown in Figure 5.1a.

Users can choose to upgrade their version of the PHP system, in which case `phc` will automatically assume the new behaviour for the generated code. However, compiled binaries may need to be re-compiled, because the language has effectively changed.

```
int main (int argc, char *argv[]) {
  php_embed_init (argc, argv);
  php_startup_module (&main_module);
  call_user_function ("__MAIN__");
  php_embed_shutdown ();
}
```

Listing 5.2: *phc generated code is called via the PHP system*

5.4.3 Run-time Code Generation

In addition to being important for correctness and reuse, the link between our generated code and the PHP system can be used to deal with PHP's dynamic features, in particular, the problem of run-time code generation.

Though the **include** statement is semantically a run-time operation, phc supports a mode in which we attempt to include files at compile-time, for performance. As the default directories to search for these files can change, we use the C API to access the include_path language flag. If we determine that we are unable to include a file, due to its unavailability at compile-time, or if the correctness of its inclusion is in doubt, we generate code to invoke the interpreter at run-time, which executes the included file. We must therefore accurately maintain the program's state in a format which the interpreter may alter at run-time. Our generated code registers functions and classes with the PHP system, and keeps variables accessible via the PHP system's local and global run-time symbol tables. This also allows us support variable-variables and the **eval** statement with little difficulty.

5.4.4 Compiling with phc

phc compiles PHP source code, generating C, as described in Section 4.1. The generated code interfaces with the PHP C API, as shown in Figure 5.1b. The generated C is compiled into an executable—or a shared library in the case of web applications—by a C compiler. Listings 5.2–5.5 show extracts of code compiled from the example in Listing 5.1. In each case, the example has been edited for brevity and readability, and we omit many low-level details from our discussion.

Listing 5.2 shows the main() method for the generated code. phc compiles all top-level code into a function called __MAIN__. All functions compiled by phc are added to the PHP system when the program starts, after which they are treated no differently from PHP library functions. To run the compiled program, we simply start the PHP

```
zval* p_i;
php_hash_find (LOCAL_ST, "i", 5863374, p_i);
php_destruct (p_i);
php_allocate (p_i);
ZVAL_LONG (*p_i, 0);
```

Listing 5.3: *phc generated code for* `$i = 0`

```
static php_fcall_info fgc_info;
php_fcall_info_init ("file_get_contents", &fgc_info);

php_hash_find (LOCAL_ST, "f", 5863275, &fgc_info.params);

php_call_function (&fgc_info);
```

Listing 5.4: *phc generated code for* `file_get_contents($f)`

system, load our compiled functions, and invoke `__MAIN__`.

Listing 5.3 shows a simple assignment. Each value in the PHP system is stored in a `zval` instance, which combines type, value and garbage-collection information. We access the `zval`s by fetching them by name from the local symbol table. We then carefully remove the old value, replacing it with the new value and type. We use the same symbol tables used within the PHP system, with the result that the source of the `zval`—whether interpreted code, libraries or compiled code—is immaterial.

Listing 5.4 shows a function call. Compiled functions are accessed identically to library or interpreted functions. The function information is fetched from the PHP system, and the parameters are fetched from the local symbol table. They are passed to the PHP system, which executes the function indirectly.

Listing 5.5 shows an `include` statement. The PHP system is used to open, parse, execute and close the file to be included. The PHP system's interpreter uses the same symbol tables, functions and values as our compiled code, so the interface is seamless.[9]

5.4.5 Optimizations

The link to the C API also allows `phc` to preform a number of optimizations, typically performing computation at compile-time, which would otherwise be computed at run-time.

[9] We note that the seamless interface requires being very careful with a `zval`'s reference count.

```
php_file_handle fh;
php_stream_open (Z_STRVAL_P (p_TLE0), &fh);
php_execute_scripts (PHP_INCLUDE, &fh);
php_stream_close (&fh);
```

Listing 5.5: *phc generated code for* `include ($TLE0)`

Constant-folding

The simplest optimization we perform is constant folding. In Listing 5.1, line 23, we would attempt to fold the constant expression `1024 * 1024` into `1048576`. PHP has five scalar types: booleans, integers, strings, reals and nulls, and 18 operators, leading to a large number of interactions which need to be accounted for and implemented. By using the PHP system at compile-time, we are able to avoid this duplicated effort, and stay compatible with changes in future versions of PHP. We note that the process of extracting the result of a constant folding does not change if the computation overflows.

Pre-hashing

We can also use the embedded PHP system to help us generate optimized code. Scripting languages generally contain powerful syntax for hashtable operations. Listing 5.1 demonstrates their use on line 20.

When optimizing our generated code, we determined that 15% of our compiled application's running time was spent looking up the symbol table and other hashtables, in particular calculating the hashed values of variable names used to index the local symbol table. However, for nearly all variable lookups, this hash value can be calculated at compile-time via the C API, removing the need to calculate the hash value at run-time. This can be seen in Listing 5.3, when the number `5863374` is the hashed value of `"i"`, used to lookup the variable `$i`. This optimization removes nearly all run-time spent calculating hash values in our benchmark.

Symbol-table Removal

In Section 5.4.3, we discussed keeping variables in PHP's run-time symbol tables. This is only necessary in the presence of run-time code generation. If we statically guarantee that a particular function never uses run-time code generation—that is to

say, in the majority of cases—we remove the local symbol table, and access variables directly in our generated code.

Pass-by-reference Optimization

PHP programs tend to make considerable use of functions written in the C API. As functions may be called which are not defined at compile-time, we must add run-time checks to determine whether parameters should be passed by reference or by copy. However, we are able to query the signatures of any function written in the C API, which allows us to calculate these at compile-time, rather than run-time.

Caching Function Calls

As PHP is a dynamic language, with functions only defined at run-time, we must lookup functions by name before we can call them. Initially, we began by looking up a function each time we called it. However, as functions cannot change their definition after they are first defined, we cache the function lookup after the first time we call it. This speedup from this optimization is significant (around 23% compared with a similar version of `phc` without this optimization).

5.4.6 Caveats

Our approach allows us to gracefully handle changes in the PHP language, standard libraries and extensions. Clearly though, it is not possible to automatically deal with large changes to the language syntax or semantics. When the parser changes—and it already has for the next major version of PHP—we are still required to adapt our compiler for the new version manually. Though we find it difficult to anticipate minor changes to the language, framing these problems to use the PHP system is generally straightforward after the fact. Finally, we are not resilient to changes to the behaviour of the C API; empirically we have noticed that this API is very stable, far more so than any of the features implemented in it. This is not assured, as bugs could creep in, but these tend to be found quickly because the API is in very heavy use, and we have experienced no problems in this regard.

5.5 Interactions with the PHP Memory Model

When assessing the performance of a programming language implementation, it is natural to think that most of the execution time is likely to be spent performing computations. In fact, as we discuss in Section 5.7.1, the run-time system often has a major impact on performance. This is particularly true for scripting languages for three main reasons. First, scripting languages generally provide automatic memory management to reclaim objects that are no longer in use. The memory manager adds to execution time, whether it uses a tracing garbage collector, or as in the case of PHP, reference counting. Secondly, even scalar values in scripting languages are typically implemented with data structures rather than simple C scalars, because additional information such as type and memory-management information must be stored along with the value. Thirdly, the main data-structuring feature provided by scripting language is the table, which is typically implemented using hashtables. Thus, even simple record or array data structures need a more complicated memory representation, which often consists of more than one single piece of memory. For these reasons, to optimize the performance of a compiler which uses the canonical implementation, it is essential to understand the memory model used by the implementation.

In this section, we discuss the *PHP memory model* and pitfalls which occur when linking to such a model.

5.5.1 PHP Memory Model

The primitive unit of data in PHP is the `zval`, a small structure encompassing a union of values—objects, arrays and scalars—and memory-management counters and flags. A PHP variable is a symbol-table entry pointing to a `zval`, and multiple variables can point to the same `zval`, using reference counting for memory management.

PHP assignment is by copy, meaning that semantically the lvalue becomes a copy of the rvalue. This is not only true of scalars: PHP arrays are deeply copied during an assignment, and object references are copied to a new run-time `zval`. As an optimization, the PHP system causes the lvalue to share the rvalue's `zval`, increasing its reference count. The variables are said to become part of the same *copy-on-write* set. Thus, even though an assignment is semantically a copy, the assigned value is shared until it is required to be altered.

Assignment can also be by reference, which puts the two variables in the same *change-on-write* set, in a similar fashion. This sets the `is_ref` flag of the shared `zval`, indicating that the variables in this set all reference each other. Setting a variable's value, where that variable is part of a change-on-write set, changes the value of all the other variables in that set.

Variables in a copy-on-write set share the same `zval`, but are not semantically related. Although this is an optimization applied by the PHP system, it is a feature which `phc` must deal with to interact with the PHP system, and so it reuses it for performance. In order to update the value of a variable in a copy-on-write set, it must first be *separated*. A copy of its `zval` is created—a deep copy in the case of arrays and strings—and the original `zval` has its reference count decremented. Variables in a change-on-write set must similarly be separated if they are assigned by copy.

Assignment to a variable in a change-on-write set overwrites the `zval`'s value field, changing the value of all the variables in that set. Variables with a reference count of one, which are in neither a copy-on-write or change-on-write set—are treated similarly.

The PHP interpreter keeps pointers to a variable's `zval` in global and function-local symbol-tables—hashtables indexed by the variable's name. When a function finishes execution, the local symbol-table is destroyed, decreasing the reference count of all `zvals` contained within. The global symbol-table is destroyed at the end of the execution of a script.

5.5.2 Pitfalls with the Memory System

In creating `phc`, we came across a number of pitfalls which we believe can affect any scripting language compiler. We describe the most important which we have come across.[10] At first, our naively generated code was around ten times slower than the PHP interpreter. This was primarily due to the fact that our code used significantly more memory than the PHP interpreter. The most important factor in this was our use of three-address-code (TAC). In order to simplify our compiler transformations and code generation, we lowered complex expressions into TAC by adding assignment to temporary variables. However, these extra assignments increase the reference count of a `zval`, meaning not only that a program's memory remains live for a longer period, but also that there are more separations, leading to extra memory allocations, copying, and subsequent deallocations.

[10]Another interesting pitfall is described by Tozawa et al. [2009].

```
for ($i = 0; $i < $N; $i++)
{


    $str .= "hello";    // concat

}
```

(a)

```
for ($i = 0; $i < $N; $i++)
{
    $T1 = "hello";
    $T2 = $str;         // T2.refcount++;
    $T2 = $T2 . $T1;    // concat
    $str = $T2;
}
```

(b)

Listing 5.6: *String concatenation benchmark*

In a simpler language such as C, copying a value has no ramifications for the copied value, so introducing TAC does not have great performance side-effects. However, in PHP, copying a value will increase its reference count, meaning it must be separated before it can be written to or altered. We removed many of the cases in which we generated poor code simply by being more careful during our conversion to TAC.

To highlight the magnitude of this problem, consider Listing 5.6a. In this example, we accidentally turn an $O(N)$ algorithm into an $O(N^2)$ one, shown in Listing 5.6b. This is a subtle, but interesting problem stemming from the interaction of TAC and copy-on-write semantics. Other scripting languages which use copy-on-write, such as Perl and Tcl, may also experience this problem.

Listing 5.6a is a string concatenation benchmark, referred to later as *strcat*. The `.=` operator performs in-place concatenation, in this case appending `"hello"` onto the end of the string in `$str`. Though this code did not strictly need to be lowered to TAC, our over-zealous lowering algorithm added extra temporaries into this code, resulting in Listing 5.6b. Semantically, these perform the same operations. However, the `zval` pointed to by `$T2` has a reference count of two after line 4, meaning the string cannot be concatenated in place. Instead, `$T2` must be separated, even though it will be freed on line 4 of the next loop iteration.

It is interesting to observe the difference in performance between the two similar pieces of code. Listing 5.6a takes $O(N)$ time.[11] By contrast, in Listing 5.6b, when `$str` must be copied in every iteration due to an increased reference count, the same work takes $O(N^2)$ time in total. We note that this problem does not only occur due to TAC. It is not always trivial to determine the reference count of a variable, and problems such as these may appear in user-code by accident.

[11] We ignore the complexity of memory allocation due to increasing the size of the string, which will be the same in both cases.

5.6 Just-in-time Compilers

Just-in-time compilers (JITs) [Aycock, 2003] are an alternative to interpreting or ahead-of-time compiling. In recent years, the growing popularity of managed languages running on virtual machines, such as Java's JVM and the Microsoft .Net framework, has contributed to the growth of JIT compilers.

JIT compilers' optimizations are not inhibited by dynamic features, such as reflection and run-time code generation. Method specialization [Rigo, 2004] compiles methods specifically for the actual run-time types and values. Other techniques can be used to gradually compile hot code paths [Gal et al., 2006, Zaleski et al., 2007].

JIT compilers, however, suffer from great implementation difficulty. They are typically not portable between different architectures, one of the great advantages of interpreters. Every modern scripting language's canonical implementation is an interpreter, and many implementations sacrifice performance for ease of implementation. The Lua Project [Ierusalimschy et al., 2005, Section 2], for example, strongly values portability, and will only use ANSI C, despite potential performance improvement from using less portable C dialects, such as using computed `goto`s in GNU C.

In addition to being difficult to retarget, JIT compilers are difficult to debug. While it can be difficult to debug generated code in an ahead-of-time compiler, it is much more difficult to debug code generated into memory, especially when the JIT compiler compiles a function multiple times, and replaces the previously generated code in memory. By contrast, our approach of generating C code using the PHP C API is generally very easy to debug, using traditional debugging techniques.

Much of the performance benefit of JIT compilers comes from inlining functions [Suganuma et al., 2002]. However, scripting language standard libraries are typically written using the language's C API, not in the language itself, and so cannot be optimized using the JIT compiler's inlining heuristics. We also expect a similar problem when current methods of trace JIT compilers [Gal et al., 2009] are attempted to be ported from Javascript—in which entire applications are written mostly in Javascript—to other scripting languages. Achieving the kind of speeds achieved by Java JIT compilers would require rewriting the libraries in the scripting language. As a result, it often takes great effort to achieve good performance in a JIT compiler. A prototype JIT compiler for PHP [Lopes, 2008] was recently developed using LLVM [Lattner and Adve, 2004], but ran 21 times slower than the existing PHP interpreter.

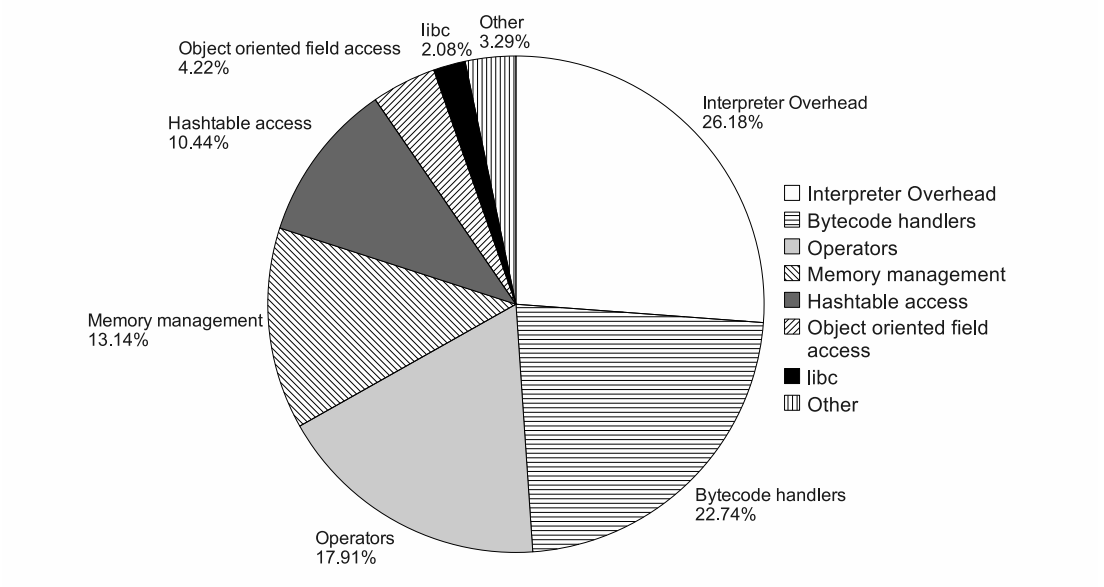


Figure 5.2: *Profiling results of the PHP interpreter, using* `callgrind`

5.7 Evaluation

5.7.1 PHP Performance Profile

Conventional wisdom states that a compiled program should run an order-of-magnitude faster than an interpreted program. In our experience, however, dynamic scripting languages do not follow this rule of thumb. Instead, a program written in a scripting language spends most of its run-time handling dynamic features, such as dynamic types and `zvals`. This limits the potential improvement of simply removing the interpreter loop. This is particularly important for a compiler like `phc` which re-uses the PHP system, as many of the code paths executed will be the same, whether the program is interpreted or compiled.

To understand where time is spent in the PHP system, and to determine the potential speedup from optimization, we profiled the PHP system. Figure 5.2 shows the profile of a number of PHP benchmark applications, interpreted using PHP version 5.2.3, using the `callgrind` tool from valgrind 3.4.1 [Nethercote and Seward, 2007]. We compiled PHP using `gcc` version 4.4.0, using the options `-O3 -g -NDEBUG`, targeting the x86-64 instruction set. We analysed the flat profile provided by `callgrind`, looking at the '*self*' results (that is, time spent in a function, not including time spent in the function's callees). We categorized each function in the profile into broad categories, based on our knowledge of the design of the PHP system.

Interpreter overhead includes time spent parsing programs, generating bytecode, running the interpreter loop and dispatching to bytecodes. *Bytecode handlers* are the code blocks dispatched to by the interpreter, which actually execute the desired operation. *Operators* includes time spent executing arithmetic and logical operators. *Memory management* is self explanatory. *Hashtable access* involves access to hashtables (which includes arrays, objects and symbol-tables), including calculating hash values from string keys. *Object-oriented field accesses* excludes the actual hashtable access, but includes other object-oriented overhead such as checking for special object-oriented handlers. *libc* denotes time spent in the C standard library.

While there is a significant amount of time spent in interpreter overhead (26%), it is not nearly enough to allow for a order-of-magnitude speedup from compilation. This lends support to our approach, as compared to that of 211 and Python2C. Both of these Python compilers take a narrow approach, attempting only to remove interpreter overhead, but they do not allow for higher-level optimizations. This means that their techniques cannot achieve a great speedup if they were applied to the PHP system.

Nearly 18% of run-time time is spent performing calculations in the *Operators* category. This is principally due to PHP's dynamic typing. PHP uses opcodes which perform significantly more computation than, say, a Java bytecode. For example, an *add* uses a single opcode, like in Java. However, where a Java add opcode is little more than a machine add and an overflow check, PHP's add opcode calls an add function. This function, depending on the types of the operands, might merge two arrays, convert strings to integers, or a number of other operations.

We also see a 10% overhead due to hashtable accesses. Hashtables are used extensively in PHP, not only as the principle data structure (as both arrays and associative arrays), but also to provide symbol-tables and objects. In theory, the PHP system's interpreter accesses every local variable through the local symbol-table. However, it uses an optimization similar to our symbol-table removal in Section 5.4.5, which prevents this overhead [Wharmby, 2006]. As a result, all of the hashtable overhead comes from array manipulation, accesses to the global symbol-table, and accessing fields of objects.

PHP's dynamic typing cross-cuts all of these categories. Hashtables must be used in PHP's object-orientation, as a result of objects' dynamic types. A great deal of memory management is due to allocating `zvals` for every value in the program, a result of dynamic typing. A lot of the overhead of operators are due to checking types before performing the computation, which might be cheap by comparison. Thus dynamic types not only take up time in the PHP system, but also prevent compiling

PHP programs to more efficient representations. We believe that static analyses can be developed which can remove many of these type checks and allow more efficient compilation. Chapter 6 discusses our efforts in this regard.

5.7.2 Performance

The major motivation of this research is to demonstrate a means of achieving correctness in a scripting language reimplementation. However, we are also able to increase the performance of our compiled code, compared to the PHP system's interpreter.

The PHP designers use a small benchmark [The PHP Group, 2007], consisting of eighteen simple functions, iterated a large number of times, to test the speed of the PHP interpreter.

We compared the generated code from `phc` with the PHP system's interpreter, version 5.2.3. We used Linux kernel version 2.6.29.2 on an Intel Xeon 5138 with four cores,[12] rated at 2.13 Ghz (clocked at 1.6 Ghz), with 12GB of RAM and a 1MB cache per CPU. Both our compiled code and the PHP system were compiled with `gcc` version 4.4.0, using `-03 -NDEBUG` compiler flags.

Figure 5.3 shows the execution time of our generated code relative to the PHP interpreter. `phc` compiled code performs faster on 16 out of 18 tests. The final column is the arithmetic mean of the speedups, showing that we have achieved a total speedup of 1.55. In Figure 5.4, our metric is memory usage, measured using the *space-time* measure of Valgrind 3.2's `massif` tool [Nethercote and Seward, 2007]. Our graph shows the per-test relative memory usage of one implementation over the other. The final column is the arithmetic mean of the reductions in memory usage, showing a reduction of 1.30.

It can be expected that we are able to optimize away the interpreter overhead, as discussed in Section 5.7.1, to achieve a speedup of 1.35. This is in the same league as previous implementations. Python2C [Salib, 2004, Section 1.3.1] is reputed to have a speedup of approximately 1.2, using a similar approach to ours, including some minor optimizations. 211 [Aycock, 1998] only achieves a speedup of 1.06, the result of removing the interpreter dispatch from the program execution, and performing some local optimizations. It removes Python's interpreter dispatch overhead, and

[12] Note that all of our benchmarks are single-threaded, and that PHP does not support threads at a language level.

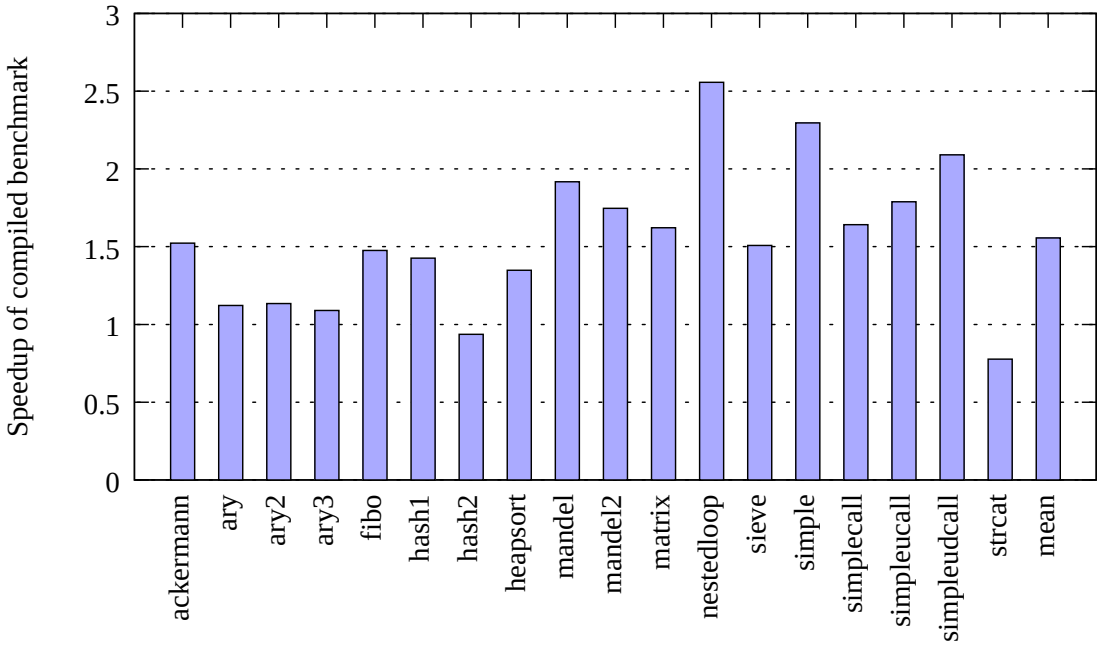


Figure 5.3: *Speedups of* phc *compiled code vs the PHP interpreter. Results greater than one indicate* phc*'s generated code is faster than the PHP interpreter. The mean bar shows* phc*'s speedup of 1.55 over the PHP interpreter.*

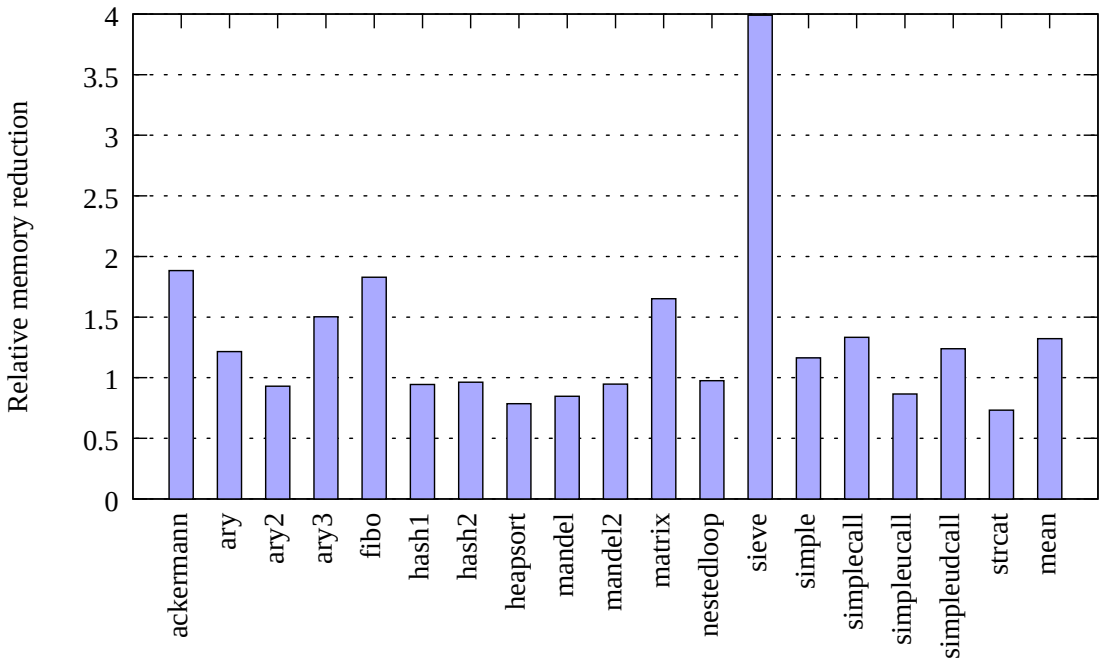


Figure 5.4: *Relative memory usage of* phc *compiled code vs the PHP interpreter. Results greater than one indicate* phc*'s generated code uses less memory than the PHP interpreter. The mean bar shows* phc*'s a memory reduction of 1.30 over the PHP interpreter.*

removes stores to the operand stack which are immediately followed by loads. We do not benefit from 211's optimization as peephole stack optimization will also not work for PHP, which does not use an operand stack.

However, our speedup is in some cases much greater than that which can be achieved by simply removing the interpreter overhead. In most cases, these are due to the optimizations which we discussed in Section 5.4.5. However, these are mitigated in some cases by poor code generation, especially related to hashtables, for example in `ary`, `ary2`, `ary3` and `hash2`. By contrast, we achieve much better speedups in functions which primarily manipulate loops and integers, in particular `nestedloop` and `mandel`.

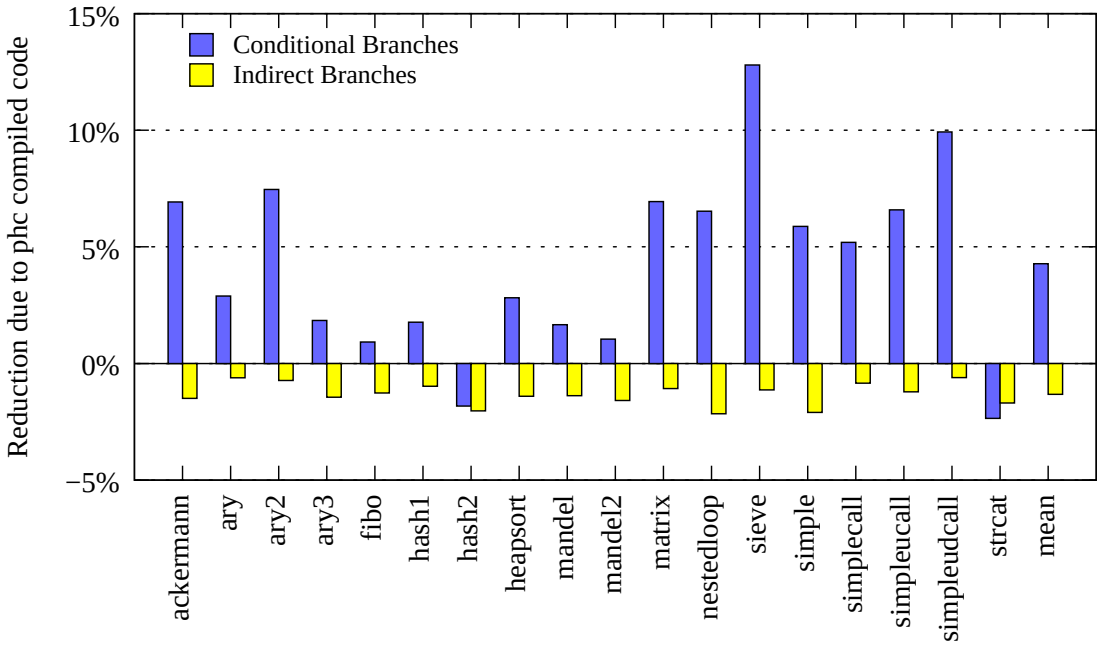
We expect that traditional data-flow optimizations will also greatly increase our performance improvement, and our approach allows this in the future, which neither 211 nor Python2C allow. We believe that without this ability, 211 and Python2C are likely dead-ends, with their performance limited by their approaches.

We also believe that the PHP system could achieve higher performance with a better implementation. However, the run-time work which slows PHP down also slows down our generated code, and so as PHP is improved, our speedup over PHP will likely remain constant.

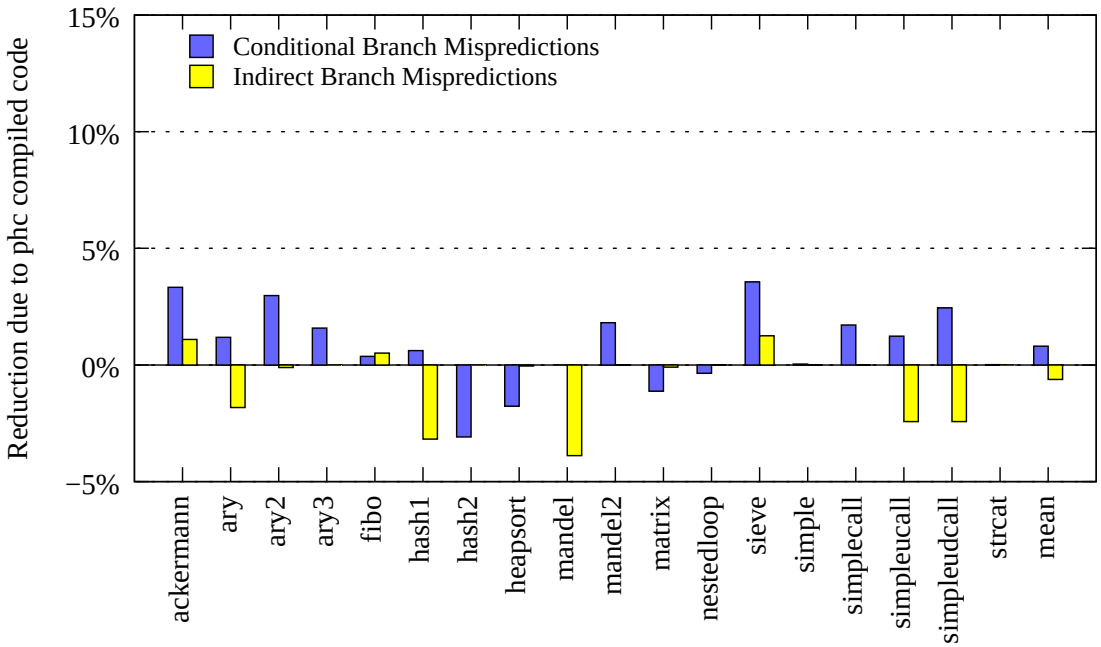
5.7.3 Performance Examination

In order to understand why we achieved our performance improvement, we analysed both interpreted and compiled PHP benchmarks using the `cachegrind` tool from Valgrind 3.4.1, a hardware simulator. We measured a wide range of metrics including instruction counts, level-1 and level-2 data and instruction cache access, and branch behaviour. We use the same benchmarks, tools and program versions as discussed in Section 5.7.2.

Figure 5.5 presents branch prediction results, with Figure 5.5a showing the change in the number of branches, and Figure 5.5b showing the change in branch mispredictions. Results above zero indicate the decrease in branches as a percentage of instructions executed in the compiled program; results below zero indicate an increase. Figure 5.5b shows the difference in branch mispredictions scaled by the approximate cost of a branch misprediction. We choose 12 as this cost factor, roughly the length of a modern pipeline.



(a) *Hardware simulation results comparing the number of branches in* phc *compiled code vs that of the PHP interpreter. Results are presented as a percentage of the instruction count. Results greater than zero indicate* phc*'s generated code executes fewer branches.*



(b) *Hardware simulation results comparing the number of branch mispredictions in* phc *compiled code vs that of the PHP interpreter. The number of mispredictions are multiplied by 12 (representing a 12-stage pipeline), and presented as a percentage of the instruction count. Results greater than zero indicate* phc*'s generated code generates fewer branch mispredictions.*

Figure 5.5: *Branch misprediction results*

A major difference between interpreters and compilers is that an interpreter loop typically leads to a great number of indirect branch mispredictions. Our results do not show this expected decrease in branch mispredictions however. Instead, we have a slight increase in indirect branches, of approximately 2%, and a larger decrease in conditional branches. In Figure 5.5b, we can see that the number of branch mispredictions does not decrease in our compiled code. It shows that even when scaled by a factor of 12, the cost of branch mispredictions costs no more than an equivalent of 1% of instructions executed. We also see that the slight increase in indirect branch mispredictions is mitigated by the slight decrease in conditional branch mispredictions.

We believe that the cost of the interpreter loop is not great in the PHP interpreter, when compared to the cost of dynamic features. Our generated code heavily uses switch statements in order to handle dynamic typing, and it appears that the reduction in the number of indirect mispredictions due to interpreter overhead is small compared mispredictions due to type checks. We speculate that the PHP interpreter more often uses conditional statements for the same purpose. Indeed, it appears that the overall number of branch mispredictions is largely the same in both compiled and interpreted programs.

We also measured changes in level-1 and level-2 cache misses, for both instruction and data caches. The difference in these misses is insignificant (that is, approaching 0%) when compared to instruction count, so we do not present them visually. We would expect to have an increase in instruction cache misses due to essentially inlining the bytecode handlers, but this did not materialize. We believe that with larger benchmarks, this may become more apparent.

It is clear that the speed of the running programs is not greatly affected by cache accesses or branch predictors. Figure 5.6 shows the decrease in instruction count and memory accesses due to compilation. As the number of cache misses is not different, we surmise that the memory accesses removed due to compilation were level-1 cache hits, which have a low cost. Nevertheless, the ebb and flow of Figure 5.6 matches that of our speedup in Figure 5.3. It seems clear that the decrease in instruction count is due somewhat to the decrease in conditional branches. Indeed, in Figure 5.5a only two benchmarks (`hash2` and `strcat`) have an increase in conditional branches, and those same benchmarks are the only ones to result in a slowdown instead of a speedup in Figure 5.3.

As a result, we believe that our speedups come not from removing the cost of mis-

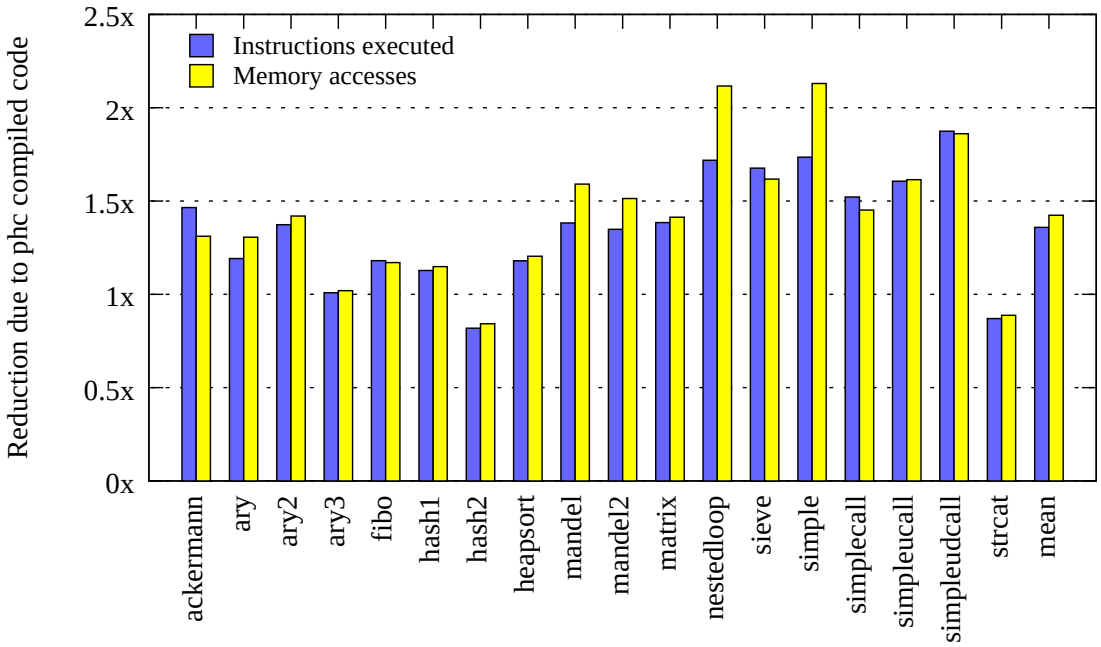


Figure 5.6: *Hardware simulation results comparing the number of executed instructions and memory accesses in* `phc` *compiled code vs that of the PHP interpreter. Results greater than one indicate* `phc`*'s generated code performs better.*

predictions in the interpreter loop, but instead through a combination of removing the rest of the interpreter overhead, and small optimizations. One of the costs of the interpreter is an extra layer of indirection when accessing `zvals`. While we store pointers to `zvals` in registers, the PHP interpreter fetches pointers to `zvals` from memory, leading to increased memory accesses. While most of our simple optimizations are local, and aimed at reducing the instruction count, removing symbol-tables is aimed at reducing memory accesses, at which we appear to have been largely successful.

5.7.4 Feedback-directed Optimization

Our technique is roughly similar to inlining the PHP system's bytecode handlers. In theory, this could allow the code to be rearranged based on feedback-directed optimization (FDO). This might allow the C compiler to do aggressive optimization, in a similar technique to speculative inlining [Feeley, 2007] or trace trees [Gal et al., 2009]. Ideally, this would mitigate the slowdown of some of PHP's dynamic features, in particular its dynamic type checks, by moving the most likely code into a straight path, eliding pipeline stalls and branch mispredictions.

In order to determine whether such profiling has a beneficial effect, we reran our

benchmarks using the gcc 4.4's FDO feature. Figure 5.7 shows the speed improvements over PHP 5.2.3, when using feedback directed optimization. PHP was configured as discussed above. We compiled phc generated code in the same manner as above, with the exception that we used the FDO options from gcc 4.4.0. We compiled the benchmarks initially using the `-fprofile-generate` flag. After running the generated executable, we compiled the benchmarks again using its feedback, with the `-fprofile-use -fprofile-generate` flags. Finally, we reused that feedback when compiling the benchmarks again using the `-fprofile-use` flag only.

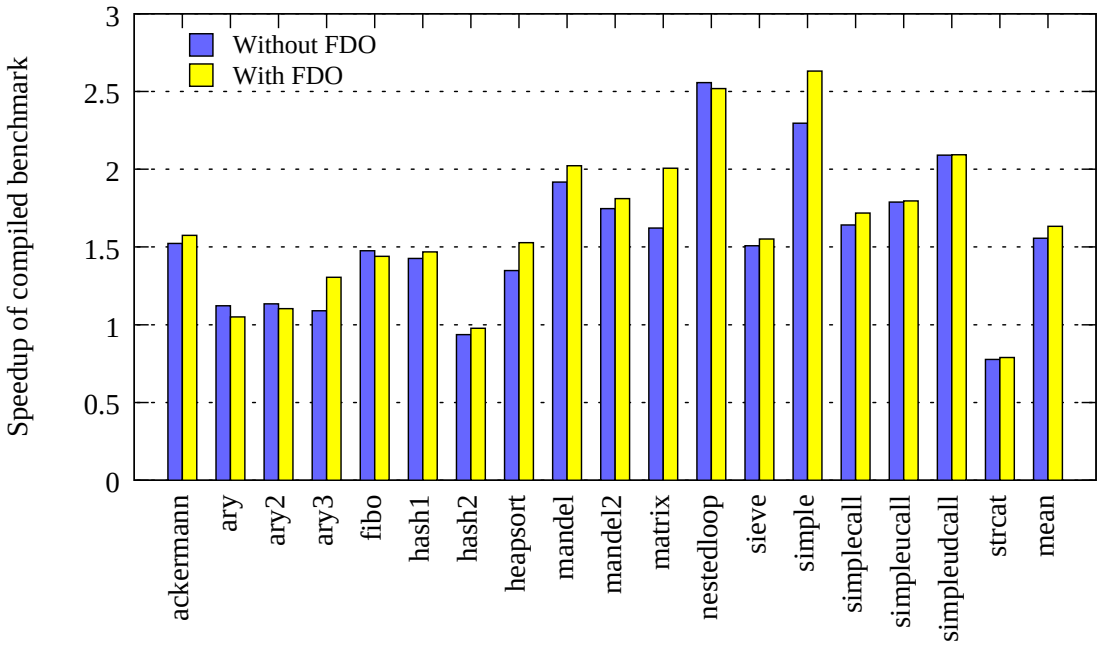


Figure 5.7: *Speedups of phc compiled code vs the PHP interpreter, with and without FDO. Results greater than one indicate phc's generated code is faster than the PHP interpreter. The* 'Without FDO' *bar repeats the results from Figure 5.3. The mean* 'With FDO' *bar shows phc's speedup of 1.63 over the PHP interpreter, when using feedback-directed optimization.*

In Figure 5.7, the '*Without FDO*' bar repeats the data from Figure 5.3. The '*With FDO*' bar shows the speedup over the PHP interpreter, when the code is compiled using FDO. Note that neither the PHP interpreter, nor the PHP system, are compiled using FDO.

It seems that while we achieve a small speedup from FDO, we are not able to automatically achieve large speedups. FDO causes our speedup to increase from 1.55 to 1.63. Most of the results indicate a small speedup, with the occasional small slowdown. While this average speedup is not insignificant, it is clear than most of the changes we seek can not be done at such a low-level, but will instead have to be handled within phc. In the future, we will attempt to incorporate FDO within phc, applying a technique

similar to that of Feeley [2007].

Currently FDO provides a small speedup which is not possible in an interpreted environment. Our generated code separates the bytecode handlers' code paths in a context-sensitive manner. As the C code is essentially inlined, it can be optimized using the profile for a single application. Naturally, we link the compiled code to the PHP system, which is not optimized in this way. However, we are still able to automatically achieve a small improvement by exposing `phc` generated code to the C compiler.

This optimization is not reasonable for an interpreted program. Other programs may need to be executed by the same interpreter, and may not benefit from the same optimizations, due to having a different profile.

5.7.5 Run-time Code Generation in PHP Programs

The techniques we describe in this chapter are particularly useful in the presence of run-time code generation. To evaluate its utility, we attempted to determine how often run-time code generation was used, by analysing a large number of publicly available PHP programs.

We automatically downloaded source code packages from the open-source code hosting site `sourceforge.net`. We selected packages which were labelled with the tag *php* and contained PHP source files. Of 645 packages chosen automatically, 581 of them contained an **`include`** statement. We consider these our test corpus, excluding packages without a single **`include`** statement. We believe files without **`include`** statements are likely to be simple programs or small classes, and are unlikely to be complete PHP programs. Figure 5.1 show overall statistics for the analysed code, showing we analysed over 42,000 files, incorporating over 8 million lines of code.[13]

	PHP files	SLOC	**`includes`**
Total	42,523	8,130,837	66,999
Average	73	13,995	115

Table 5.1: *Package statistics for 581 PHP code packages, including number of files, number of source lines of code (SLOC), and number of* **`include`** *statements.* **`include`** *statements also includes* **`require`**, **`include_once`** *and* **`require_once`** *statements. 'Average' means per package.*

We created a plugin for the `phc` front-end to determine the presence of run-time code

[13] We measured lines of code using the Unix utility `wc`, so this figure includes blank lines and comments.

generation. We searched for either `eval` statements, or `include` statements which used dynamic features. We considered `include` statements which used only PHP constants, literal strings and concatenations to be static—all other features were deemed to be indicative of run-time code generation. We show the results of this analysis in Figure 5.2.

	Dynamic `include`s	`eval`s	Either	Neither
Instances	11,731	1,586		
Packages	331 (57%)	156 (26.9%)	358 (61.6%)	223 (38.4%)
Average	35.4	10.2		

Table 5.2: *Dynamic features in PHP code. The rows are: the number of instances of each feature, the number of packages using the feature at least once (with percentage of total packages), and the average number of times the feature is used by packages which use it.*

From these figures, it is clear that support for run-time code generation is exceptionally important. It is used in 61% of PHP application, and when it is used, it is used extensively, with `eval`s appearing over 10 times in each package in which they appear, and dynamic includes appearing 35 times in each package in which they appear. This strongly indicates that our approach of supporting these features in our ahead-of-time compiler was wise, and that more static approaches would be unable to compile a large amount of PHP code. In fact, less than 39% of PHP applications do not use these dynamic features (though other dynamic features exist, which we did not attempt to detect).

Dynamic `include` statements are typically either plugin mechanisms or provide localisation. We suspect that in many cases, localization could be handled statically. This would mean searching for files in the source directories and replacing the dynamic include with a switch statement and a set of static includes. This approach is used in other tools [Wassermann and Su, 2007]. However, it is not safe, as the directory in which to search can be altered at run-time.

While dynamic `include` statements are prevalent, and require special support, we note that the large majority of `include` statements use a static string. Of the 66,999 includes, fewer than 18% of them are dynamic. This implies that static analysis of PHP can be useful in a lot of cases, if code generation is not required.

5.8 Conclusion

Scripting languages are becoming increasingly popular, however, existing approaches to compiling and reimplementing scripting languages are insufficient. We present techniques used by phc, which effectively support three important scripting language features which have been poorly supported in existing approaches. In particular, we effectively handle run-time code generation, the undefined and changing semantics of scripting languages, and the built-in C API.

A principle problem of compiling scripting languages is the lack of language definition or semantics. We believe we are the first to systematically evaluate linking an interpreter—our source language's de facto specification—into our compiler, making it resilient to changes in the PHP language. We describe how linking to the PHP system helps to keep our compiler semantically equivalent to PHP, which has previously changed between minor versions.

We also generate code which interfaces with the PHP system. This allows us to reuse not only the entire PHP standard library, but also to invoke the system's interpreter to handle source code generated at run-time. We discuss how this allows us to reuse built-in functions for PHP's operators, replicating their frequently unusual semantics, and allowing us to automatically support those semantics as they change between releases. Changes to the standard libraries and to extensions are also supported with this mechanism.

Through discussing existing approaches, we show that our technique handles the difficulties of compiler scripting languages better than the existing alternatives. We show too that the percentage of PHP packages which benefit from our approach exceeds 60% of our sample. We show that we achieve a speedup of 1.55 over the existing canonical implementation, and present a detailed discussion of this result.

Overall, we have shown that our approach is novel, worthwhile, and gracefully deals with a number of significant problems in compiling scripting languages, while maintaining semantic equivalence with the language's canonical implementation. We believe in the importance of correctness when compiling scripting languages, and that our research will provide the stepping stone on which future optimizations can be based.

The next chapter describes the static analysis framework which can be used as the first step in these optimizations.

Static Analysis of Dynamic Scripting Languages

Static analysis is an important tool for detecting security flaws, finding bugs, and improving compilation of programs. However, static analysis of scripting languages such as PHP is difficult due to the features found in these languages. These features include run-time code generation, dynamic typing with implicit type conversions, dynamic aliasing, implicit object and array creation, and overloading of simple operators. We find that as a result, simple analysis techniques such as SSA and def-use chains are not straightforward to use, and that a single unconstrained variable can ruin our analysis. In this chapter we describe the challenging semantics of PHP, and a static analyser to model them. In particular our analysis combines alias analysis, type-inference and constant-propagation for PHP, computing results that are essential for other analyses and optimizations. Our analysis is specifically designed for PHP, and differs significantly from previous analyses, whose flaws we highlight. Our empirical results show that almost all program variables are unaliased, and that over 60% of dynamic types are statically known. Ours is the first *conservative* static analysis which allows the generation of useful results for PHP.

6.1 Motivation

As scripting languages are increasingly used for more ambitious projects, software tools to support these languages become more important. Static analysis is an important technique that is widely used in software tools for program understanding, detecting errors and security flaws, code refactoring, and compilation. However, PHP presents unique challenges to this analysis. In addition to highly dynamic features, simple statements can have hidden semantics which must be modelled in order to allow conservative program analysis. It is dynamically typed, and programs provide no type- or alias-information to the analysis writer. Its alias behaviour in particular cannot be conservatively analysed with known techniques.

PHP's challenging features include:

1. run-time source inclusion,
2. run-time code evaluation,
3. dynamic, latent typing, with implicit type conversions,
4. duck-typed objects,
5. implicit object and array creation,
6. run-time aliasing,
7. run-time access to symbol-tables,
8. overloading of simple operations.

Some of these features have been handled by earlier work. (1) and (2) have been addressed by string analysis [Wassermann and Su, 2007] and profiling [Furr et al., 2009], early work has been performed on alias analysis (6) [Jovanovic et al., 2006], and a number of other problems (3, 8) have also been touched upon [Xie and Aiken, 2006]. Static analyses of other scripting languages have been performed, notably for Javascript [Jang and Choe, 2009, Jensen et al., 2009], which have solved some of the same problems (3, 4). However, large sections of the PHP language are left unmodelled by this previous work, including important cases relating to arrays, and essential features such as object-orientation.

In particular, it is notable that all previous analyses for PHP have been non-conservative. They have been aimed at bug-finding, but their solutions have not been suitable for purposes such as compilation or optimization. This may be due to the difficulty of providing a conservative analysis for PHP, which we believe we are the first to do.

PHP's feature set allows simple expressions to have hidden effects based on a value's run-time type. These hidden effects are exacerbated by PHP's run-time aliasing, and their detection is hampered by PHP's dynamic typing. Their presence makes it very difficult to determine a simple list of definitions and uses of variables within a function.

Let us consider an assumption made by earlier work [Jovanovic, 2007, Section VI.B], which does not perform type-inference or model PHP's implicit type conversions and

array instantiations[1]

> "if the expression `$a[2]` appears somewhere in the program, then `$a` certainly is an array."

Unfortunately, this was untrue for their analysis on PHP 4, and is less true in PHP 5. In a reading context, `$a[2]` may read from a string, an array, an uninitialized variable or any scalar value (in which case it would evaluate to `NULL`). In the writing context, it might write into a string, assign an array value or convert a `NULL` value or uninitialized variable into an array. If `$a` was an *int*, *real*, *resource* or *bool*, then writing to it would generate a run-time warning, and the value would be unaffected. In PHP 5, if `$a` was an object, then reading from it would call the `get` object handler, if present, or else there would be a run-time error.

It should be clear from this example that many of PHP's features make analysis difficult, and that traditional analyses are not adequate to analyse these features. Existing work on Javascript [Jang and Choe, 2009, Jensen et al., 2009] naturally do not model PHP references correctly, and use a different class/object model. Even existing work on PHP omits many important features which we model correctly.

In this chapter we present novel and elegant abstractions to enable these analyses. Our analysis is whole-program, optionally flow- and context-sensitive, and combines alias analysis, type-inference and constant-propagation. It is an ambitious analysis which models almost the entire PHP language. In particular, we handle each of (3)–(8), most of which have never even been discussed in a program analysis context. We believe we are the first to handle such a large portion of the PHP language, as we are able to analyse a large number of features which were not handled by previous analyses.

In Section 6.2, we present the semantics of PHP from the perspective of program analysis, particularly discussing features which make analysis difficult. In Section 6.3 we discuss previous analyses for PHP, and how their limitations prevent analyses of the whole PHP language. We focus in particular on areas where the unconventional semantics of PHP have led to edge cases not modelled by previous work. Section 6.4 describes how traditional analyses for C or Java are insufficient to model PHP. In Section 6.5 we describe our novel analysis, how it solves the problems highlighted in Section 6.2, and the deficiencies in all previous work to date. Our experimental evaluation and discussion is presented in Section 6.6.

[1] In PHP syntax, `$a[2]` indexes the value in the variable $a.

6.2 PHP Language Behaviour

PHP has no formal semantics, no rigorous test suite, and an incomplete manual. As such, there is no definition of its semantics, except for its canonical implementation. In Chapters 4 and 5 we described a technique for creating a compiler for such a language.

Although PHP is very different from languages such as C, C++ and Java, there are a great deal of similarities to other scripting languages such as such as Javascript, Lua, Perl, Python and Ruby.

PHP offers many language features which makes program analysis difficult. In order to prime the description of our analysis in Section 6.5, we present first an informal description of the behaviour of PHP 5. We have omitted behaviour which has no effect on program analysis.

6.2.1 PHP Overview

PHP has evolved since its creation in 1995. It was originally created as a domain specific language for templating HTML pages, which was implemented in Perl.

It is influenced by its Perl roots, and has similar syntax and semantics, including a latent, dynamic type system with implicit type conversions, powerful support for hashtables, and garbage collection. PHP programs are typically web applications, and PHP is tightly integrated with the web environment, including extensive support for database, HTTP and string operations.

PHP 3 introduced simple class-based object-orientation, which used hashtables in its implementation. As tables are copied during assignment, syntax was introduced to allow variables to reference other variables, so as to pass arrays and objects to functions and methods.

PHP 5 introduced a new object model, with new assignment semantics for objects. This allowed object pointers to be copied into new values, creating a means of passing objects by pointer value. However, the old reference semantics remain, and are still required for passing arrays or scalars by reference.

6.2.2 Dynamic Typing

Typically, in statically typed imperative languages such as C, C++ and Java, names are given to memory locations. Variables, fields and parameters all have names, which typically correspond to space on the stack or the heap. Each of these names is annotated with type information, and this is statically checked.

PHP is instead dynamically typed, meaning type information is part of a run-time value. The same variable or field may refer to multiple run-time values over the course of a program's execution, and the types of these values may differ.

6.2.3 Arrays

Arrays in PHP are maps from integers or strings to values. The same array can be indexed by both string and integer indices. Arrays are not objects, and cannot have methods called on them. They are implemented as hashtables in the reference implementation, and we refer to them as *tables* from here-on-in. Tables form the basis of a number of structures in PHP, including objects and symbol-tables. Arrays are copied using a deep copy.[2]

6.2.4 Symbol-tables

PHP variables have subtle differences to languages such as C, where they represent stack slots. Instead PHP variables are fields in global or local symbol-tables, which form a map of strings to values.

This important feature bears repeating. All PHP values exist on the heap. Variables are merely names that refer to these values. The same values may be referred to by array indices or object fields as well as by variables. As we discuss in Section 6.2.11, a value may be referred to by a number of different names in the program at the same time.

PHP provides run-time access to symbol-tables. The `$GLOBALS` variable provides access to any global variable by name. The local symbol-table can be accessed dynamically

[2] The PHP reference implementation uses copy-on-write to prevent the cost of these operations, but the semantics are still to copy. Tozawa et al. [2009] present a problem with this implementation, and a solution. Copy-on-write does not affect our analysis, and so we do not discuss it further.

```
$x = "old value";
$name = "x";
$$name = "new value";
print $x;       // "new value"
```

Listing 6.1: *Example of the use of variable-variables*

using *variable-variables*. A variable-variable provides a means of reading or writing a variable whose name might only be known at run-time. Listing 6.1 demonstrates reading to and writing from the variable `$x`. Variable-variables make it clear that variables are just symbol-table entries: it is possible to read and write to variables which would violate PHP's syntax check using variable-variables, such as variables named '*+*' or '*#@$!*'.

PHP's built-in functions occasionally have access to the symbol-table of their callers. The built-in functions **`compact`** and **`extract`** respectively read and write from the caller's symbol-table, bundling and unbundling values from variables into arrays. The variables that are read and written-to can be chosen based on dynamic values. A number of other functions also write the the `$php_errormsg` variable in case of error.

The `$GLOBALS` variable, which points to the global symbol-table,[3] can be passed to array functions, iterated over, or cleared. In fact, the user can even unset the `$GLOBALS` variable, possibly preventing direct access thereafter.

Many of the behaviours of arrays are shared by other features which are implemented using arrays. For example, reading from an uninitialized field in a PHP array will return `NULL`, and so will reading an uninitialized value from a symbol-table as a result. These behaviours are also shared by PHP's objects.

6.2.5 Objects

PHP's class system is a mishmash of other type systems. It was in principle copied from Java, and then integrated into the existing type system.

Unlike a number of other scripting languages, there is a static class hierarchy. Once a class is declared its parent class cannot be changed, a fact which simplifies analysis. In addition, classes may not change their methods after they are declared. Similarly, once an object is instantiated it cannot change its class, and as a result its methods may not be changed either.

[3] which holds the `$GLOBALS` variable...

```
1 class MyInt {
2     function value () {
3         $this->num = rand ();
4         return $this->num;
5     }
6 }
7
8 $int = new MyInt(5);
9 $newint = $int->value ();
```

Listing 6.2: *Example of field use without declaration*

Otherwise, PHP uses *duck-typing*, discussed in Section 2.1.3. Fields have no declared type, and may be added and deleted from an object at any time. As such a class does not denote a particular memory layout. Like symbol-tables, the canonical implementation implements objects using tables, and the semantics reflect that. For example, an uninitialized field (that is, accessing a field of an object which does not yet have that field) evaluates to `NULL`, as do fields of arrays and symbol-tables. However, unlike arrays, PHP objects are not deep-copied—a pointer to the object is copied as in Java.

A simple class declaration and example is shown in Listing 6.2. The field `num` is not declared anywhere. The object instantiated on line 8 does not have any fields. The `num` field is only added to the object in the `value` method on line 3, after the method is called from line 9.

Unlike arrays and symbol-tables, objects can have *magic methods*. Magic methods are methods which are implicitly triggered by invoking specific operations on objects who's class defines the magic method. Figure 6.1a contains a selection of allowed magic methods. `__get`, `__set` and `__isset` are called if inaccessible fields (missing or having private inheritance) are accessed, `__call` is called if inaccessible methods are accessed and `__invoke` is called upon an attempt to *invoke the object*.[4]

The `__toString` method is of particular interest. Strings are fundamental to PHP, and there are a large number of places in which the `__toString` method may be called, should an object be passed instead of a string. Examples are the concatenation operator ('.'), interpolated variables in strings (e.g. `"My name is $name"`), printing, or using any of PHP's built-in functions which expect strings. If you pass an object with a

[4] Variable-methods allow arbitrary values to be invoked (e.g. `$myfunc()` or `$obj->$mymethod()`). Typically, the invoked value is a string, and the name of the string is used to lookup a global function or class method. Objects with the `__call` method can be used to simulate first-class functions or closures.

__call
__get
__invoke
__toString
__set
__isset

(a) Magic methods

call_method
cast_object
get
get_constructor
get_method
has_dimension
has_property
read_dimension
read_property
set
write_dimension
write_property

(b) *C object handlers*

Figure 6.1: *Selection of magic methods and object handlers. These methods are called implicitly when certain operations are performed on an object.*

`__toString` method to a library expecting a string, the `__toString` may be called when the string is evaluated. This may happen deep within the library.

As described in Chapter 5, many PHP libraries are implemented using PHP's C API. Objects passed to or from C libraries may be given special *C object handlers*. These are implicitly triggered by simple operations such as reading an object's value or accessing it using array syntax. Figure 6.1b contains most of these handlers. If defined, they typically replace the standard behaviour of some PHP construct, for example accessing an object using the array syntax (`*_dimension`) or accessing fields (`*_property`).

As the libraries written using the C API may have access to a large portion of the internals of the reference implementation, C object handlers may seriously change the semantics of the program. For example, they may add new variables to the local or global symbol-tables, change the class of an object (which is otherwise forbidden), or rename global functions at run-time. We have seen examples of each of these behaviours in PHP libraries.

6.2.6 Operators

There are two equality operators, `==` and `===`. The former uses may apply implicit type conversions, considering for example the integer `5` to be equal to the string `"5"`. The `===` operator is stricter, requiring the type of the values to also be equal.

```
print "zest" == 0;          // true
print 0 == "eggs";          // true
print "zest" == "eggs";     // false
```

Listing 6.3: *Example of intransitive equality operations. All PHP strings are considered equal to the integer* `0`.

PHP has detailed rules on the equality of values in the presence of implicit type conversions, described in the PHP manual. Many of these rules were collected by experimentation on the existing behaviour in some version of PHP, but they appear to be unchanged since their behaviour was first added to the manual. An interesting property of the `==` operator is that it is not transitive, as can be seen in Listing 6.3. Other comparison operators, including arithmetic and bitwise operators, have similar quirks.

6.2.7 Scalar Values

PHP has the following scalar types: *int*, *real*, *string*, *bool*, *resource* and *null*. It is not possible to add user-defined scalar types.

The *int* and *real* types are wrappers for the C types *long* and *double*. The *bool* type represents simple boolean values which may be **`true`** or **`false`**. A *string* is a scalar in PHP, as there is no character type for it to aggregate. However, array syntax may be used to modify portions of a string, complicating analysis of arrays.

Much of PHP's standard library is written in C. The *resource* type provides a way to wrap C pointers in user-code, after they are returned from standard libraries. PHP user code cannot create resources, and user-space operations on them are largely not meaningful. They are only stored in values so that they may be passed back to C library functions. Resources are trivial to model in our analysis, and so shall not be addressed further.

PHP's `NULL` value closely resembles the unit value [Pierce, 2002] in Scheme. Scheme's unit type has exactly one value, also called unit. Similarly, PHP's NULL type has exactly one value, also called `NULL`. However, comparisons between PHP's `NULL` and those of other languages are muddied by PHP's implicit type conversions. For example, `NULL` is equal (using `==`) to `0`, `""`, `0.0`, and **`false`**.

PHP allows definitions of constant values, similar to C's **`#define`** construct. However, *constants* are dynamically defined in PHP, by calls to the **`define`** function. As such,

```
if (25 <= $_GET["age"])
    echo "...";
```

Listing 6.4: *Example of implicit type conversions.* `$_GET` *is a table containing values from the user, provided via a HTTP form. The user value will be implicitly converted to an integer, and compared to 25.*

they are defined at run-time, not compile-time. Although a constant may not change its value once it is defined, a constant may be conditionally defined, may be defined late in the program, or may be defined from a user value. Values of constants may depend on program control-flow. Constants may only be defined to scalar values.

6.2.8 Implicit Type Conversions

A major feature of PHP's type system is that as well as being dynamically typed, it allows implicit type conversions.[5] That is, conversions between types often happen automatically and behind the scenes. For example, strings which represent integers, such as `"25"`, may be implicitly converted to integers if used in an integer context like arithmetic. Most values can be converted to some form of string if printed or concatenated.

This is due to PHP's heritage, because it was originally designed as a language for creating web applications. At the time, it was useful to take strings directly from the user, as shown in Listing 6.4 and use them directly as values of another type. PHP best practices dictate sanitizing all user values because of potential string injection vulnerabilities, and so this practice is no longer common. However, they are still an integral part of the language.

6.2.9 Type-coercion

PHP values can also be cast from one type to another explicitly by the programmer. The PHP manual [The PHP Documentation Group, 1997-2009] goes into great detail about the behaviour of casts. As such, we will only focus on those that pertain to program analysis.

Casting a scalar to an array creates a new array containing that scalar (using the key

[5] This is sometimes referred to as weak-typing, but that term is heavily overloaded. The PHP community refers to this as type juggling.

```
                                    int *y = malloc(sizeof(int));
                                    int *x = malloc(sizeof(int));
$x = 5;                             *x = 5;
$y = $x;                            *y = *x;
$y = 7;                             *y = 7;
```

(a) (b)

Listing 6.5: *Assignment by copy, with comparable C code*

```
1
2  $x = 5;                          int *x = malloc(sizeof(int));
3  $y =& $x;                        *x = 5;
4  $y = 7;                          int *y = x;
                                    *y = 7;
```

(a) (b)

Listing 6.6: *Assignment by reference, with comparable C code*

`"scalar"`). A cast to an object is similar, except that the result is an object which is an instance of '*stdClass*'. The exception to these rules is the `NULL` value, which is cast to an empty structure. It is not possible to cast between object types (or otherwise change the concrete type of an object), or from a scalar to a particular object type.

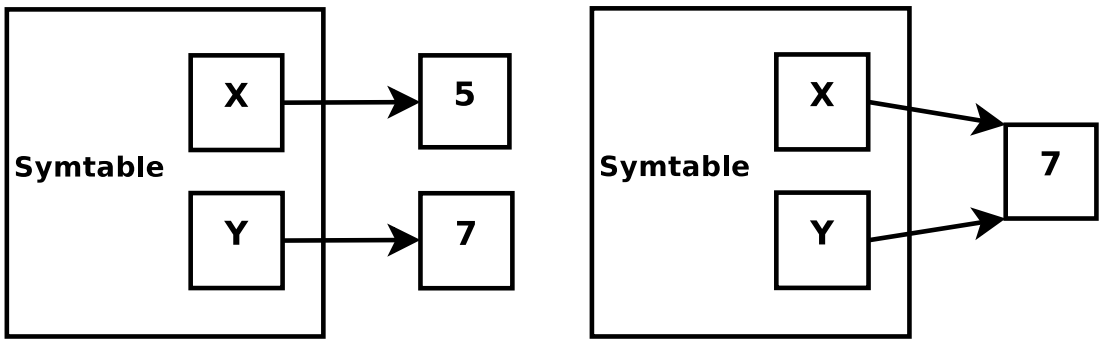
Casting an object to a string will invoke the object's `__toString` method, as discussed in Section 6.2.5.

6.2.10 Assignment

There are two forms of assignment in PHP. The most straightforward, denoted `$a = $b`, is an assignment by copy. After the assignment, `$a` holds a new copy of `$b`'s contents. In Listing 6.5, `$x` is assigned the value 5, after which `$x`'s value is copied into `$y`. Consider the run-time memory representation after the copy, shown in Figure 6.2a. As explained in Section 6.2.4, `$x` and `$y` are separate symbol-table entries, pointing to separate values.

6.2.11 References

As we explained in Section 6.2.4, PHP symbol-tables are maps from strings to values. Variables are simple keys which index symbol-tables at run-time. The same value



(a) *Assignment by copy (see Listing 6.5)* **(b)** *Assignment by reference (see Listing 6.6)*

Figure 6.2: *PHP assignment memory representation*

```
1  function a() {
2      $a2 =& $GLOBALS['x1'];
3      if (...)
4          $a1 =& $a2;
5
6      b(&$a1, $a2);
7  }
8
9  function b($fp1, &$fp2) { ... }
10
11 a();
```

Listing 6.7: *Example of dynamic aliasing in PHP. References creation can be at run-time, and in some cases occurs conditionally. This is a combination of Figures 7 and 8 from Jovanovic et al. [2006].*

may be referred to from two different names at once. It is said that these two names *reference* each other. A simple example is that a variable and an object's field may refer to the same integer value.

Assignment *by reference* creates a reference between the two variables, causing them to share a single value. In Listing 6.6, `$x`'s value is shared by `$y` on line 3, and both `$x` and `$y` are defined on line 4. Figure 6.2b shows the memory representation after line 4. In contrast to an assignment by copy, there is only a single value shared by both symbol-table entries.

Superficially, PHP references resembles references in C++, but there are important differences. PHP's references are mutable—they can be created, used, and destroyed again at run-time. Variables involved in a reference relationship do not need to be of a specific type, and there may not be syntactic clues that references are being used.

Listing 6.7 demonstrates PHP's dynamic aliasing. On line 2, `$a2` becomes a reference of `$GLOBALS['x1']`, that is, the global variable `$x1`. Line 4 shows the creation of a possible reference.

To complicate matters further, a caller may optionally pass its arguments by reference, even if the parameter is not declared as a reference parameter. Thus, either the callee or the caller can specify that a parameter is passed by reference. Line 6 of Listing 6.7 shows a call to the function `b`, where the first parameter is passed by reference in the caller, and the second parameter is passed by reference due to the signature on line 9.

When a variable is in a reference relationship, a reference assignment to it removes its old relationship. All variables in a reference relationship refer to a single value; assigning to one variable (using assignment-by-copy) updates them all. A PHP reference is bidirectional, as there is only one value being referred to by multiple names. These multiple names may be passed widely throughout the program, and the same values may be referenced by many names, including function parameters, or field of objects or arrays, or symbol-tables entries.

The difference between simple assignments in PHP and those in C or C++ are apparent in Listing 6.5. A simple assignment in PHP (`$x = 5`) more closely resembles a C assignment through a pointer (`*x = 5`) than a C scalar assignment (`x = 5`). As such, it is clear that simple PHP assignments have an extra layer of indirection, adding complexity to even simple program constructs, when compared to more static languages such as C or C++.

References and pointers[6] are not similar, except that they are both forms of aliasing. Pointers are PHP values which point to a heap object; they behave exactly like pointers in Java. They may be copied, but their representation is opaque: pointer-arithmetic is not allowed.

Two PHP values may alias by pointing to the same object. By contrast, two variables which reference *are the same PHP value*. References simply provide facilities for having a single value with multiple names (variables, fields or array-offsets).

PHP references are an important concept for program analysis. Without references, the only means of indirection in PHP would be pointers, which we must also model. One of the unique features of our alias analysis is that it handles reference relationships correctly, unlike previous analysis, as described in Section 6.3.

[6] Confusingly, PHP documentation refers to pointers as *object references*, as does Java. We avoid this term in this dissertation.

```
1 for ($i = 0; $i < 10; $i++)
2 {
3     $arr[$i] = $i;
4 }
5 $b = NULL;
6 $a =& $b[0];
```

Listing 6.8: *Examples of implicitly created values*

6.2.12 Implicit Value Creation

Certain assignment statements may have implicit effects, due to PHP's implicit type conversions. In Listing 6.8, `$arr` is uninitialized until line 3, when the assignment to `$arr[0]` converts `$arr` into an array. Assignments using object field syntax convert uninitialized values to objects of the class '*stdClass*'.

Arrays and objects are also implicitly created using references. An uninitialized variable is initialized to `NULL` if it is on the RHS of a reference assignment. Referencing fields of arrays or objects likewise initialize the field. If the array or object itself is not initialized, it too will be created. For example, `$b[0]` is initialized to `NULL` on line 6 of Listing 6.8. Due to the initialization of `$b[0]`, `$b` is automatically converted into an array. If `$b` had not already been initialized, it would have been initialized there.

Though this may seem like a minor detail, it is a motivating factor as to why type analysis is required to be integrated with alias analysis. Section 6.5.1 discusses how this affects our analysis in more detail.

6.2.13 Dynamic Code Generation

PHP has an **`eval`** statement which evaluates strings at run-time. The strings may not be known at compile-time, and may even be read from the user. Our analysis expects all code to be statically known, and so we do not deal with this in our analysis. A number of techniques [Furr et al., 2009, Wassermann and Su, 2007] exist to mitigate the pessimistic effects of **`eval`**s in static analyses.

6.3 Existing PHP Static Analyses

There has been a significant amount of research into security analysis for PHP. However, none of these analyses are conservative. That is, they make optimistic assumptions about the program in order to report possible security weaknesses to the programmer. These analyses do not need to be conservative because they only report suspicious code, and false positives are acceptable. Our analysis is conservative because we wish to safely optimize PHP programs.

More importantly, each of these analyses have incorrectly modelled portions of PHP. In this section, we discuss previous analyses for PHP, particularly highlighting their shortcomings in modelling PHP's features. We compare these analyses to our novel analysis in Section 6.7, and discuss other analyses for scripting languages in Section 3.6.

6.3.1 WebSSARI

Huang et al. [2004] performed the earliest work of which we are aware on the analysis of PHP, using WebSSARI. However, we believe it was a very early prototype which does not attempt to model PHP very well. Xie and Aiken [2006, Section 5.1] discuss the limitations of Huang's work in some detail, noting especially that it is intraprocedural and models only static types. As a result, we do not discuss it further.

6.3.2 Web Vulnerabilities

Xie and Aiken [2006] model a great deal of the simple semantics of PHP 5, with the intention of detecting SQL injection vulnerabilities. They model the **`extract`** function, automatic conversion of some scalar types, uninitialized variables, simple tables, and **`include`** statements. However, they do not discuss PHP's complex object model or references.

We note a simple misunderstanding of PHP's semantics in Xie and Aiken's [2006] work. In Section 3.1.4, they present the statement:

```
$hash = $_POST;
```

which they claim creates a symbol-table alias of `$POST` in `$hash`. In fact this is a copy of the array in `$_POST`. It is notable that earlier work has misunderstood the semantics

of PHP, which demonstrates the difficulty of analysing such a complex and under-specified language.

6.3.3 Pixy

Pixy [Jovanovic et al., 2006] provides an alias analysis for PHP 4. They correctly identify that PHP's reference semantics are difficult to model, and that ignoring this feature can lead to errors in program analysis. Their analysis is strongly focused on fixing this flaw.

A contribution of their work is the realization that previous alias analyses [Burke et al., 1994, Emami et al., 1994, Landi and Ryder, 1992] for Java, C or C++ are unsuitable for PHP. In particular, PHP's references are mutable, and are not part of the static type of a variable.

Pixy's major contribution is that they model PHP's run-time references between variables in different symbol-tables. At each point in the program, they keep track of which variables may- or must-alias each other using alias-pairs. This includes aliases between variables in the global symbol-table, other symbol-tables, and formal parameters.

There are some limitations with their approach. To begin with, it is rather complicated, with different rules for aliases between two globals, between two formal parameters, between a global variable and a parameter, and between global variables and function-local variables.

More importantly, the alias analysis in Pixy does not go far enough. They do not model that aliases may also exist between variables and members of an array.[7] Unfortunately, this breaks their *shadow* model [Jovanovic et al., 2006, Section 4.4.4] which they use to handle interprocedural analysis.

In Section 6.1, we highlighted shortcomings in Pixy's lack of type-inference. Pixy inferred that array syntax implied the presence of an array, which was not correct. In addition, they did not propagate the information they did gain, which they refer to as the problem of '*hidden arrays*'.

Finally, Pixy does not model a number of PHP 4 features:

[7] or an object's fields, but Pixy does not model objects.

- though they are aware of variable-variables, they do not mention a way to model them,
- they cannot model assignments in which the `$GLOBALS` variable is redefined. The `$GLOBALS` variable contains all variables in global scope, and redefining it changes the value of all global variables.
- they do not model functions which affect the local symbol-table (**`extract`** and **`compact`**),
- they do not model PHP 4's object-oriented features.

As their work is based on PHP 4, they also do not model PHP 5's new object model.

6.3.4 SQL Injection Attacks

Wassermann and Su [2007] performed a security analysis on PHP programs, aimed at detecting SQL injection vulnerabilities. Section 5.2 in particular details a number of limitations that led to false positives. This included incomplete support for references, and not tracking type conversions among scalar variables. This is similar to the limitations the modelling of arrays in Pixy.

Furthermore, Wassermann and Su [2007, Section 3.1.1] mention that SSA form is used. However, as there are a number of features in PHP which can touch the local or global symbol-table, it is difficult to obtain a conservative set of definitions and uses for a function. In fact, as we show in Section 6.5.5, an advanced alias-analysis is required to build an SSA form. However, As Wassermann and Su's analysis is not conservative, they do not require a conservatively correct version SSA form.

6.4 Unsuitability of Alias Analysis Models for Static Languages

PHP is a very different language to Java, C or C++, especially in terms of how it must be analysed. Although many alias analyses for C and C++, and pointer analyses for Java, have been developed, they are not suitable for analysing PHP programs.

A traditional alias analysis for Java, C or C++ typically deals with memory locations. For example the *points-to abstraction* [Emami et al., 1994] computes the points-to relation between different stack locations. This is used to model variables which are modified or used by a statement (so-called *mod-ref analysis* [Landi and Ryder, 1992] in C) to allow accurate intraprocedural scalar optimizations.

Modelling stack locations is not correct for PHP, because the values used by local variables may be shared between them, or may also be held in arrays. Obviously, there is already an implicit layer of indirection for even simple program properties.

Java has simpler object semantics than PHP. All variables and fields are scalars, and some values may be pointers, however pointers to scalar values or to values on the stack are not permitted. Java does not have references, and is therefore significantly simpler to model than PHP.

Listing 6.9 demonstrates references in C++. In it, we see a function *caller* which takes a parameter by reference. Its formal parameter `fp` is of type *int-reference*.

```
void x (int& fp);
```

Listing 6.9: *Example of C++ references*

Throughout the lifetime of `fp`, it is known that:

1. `fp` is a reference,
2. `fp` references a variable which is accessible via a different name.

More importantly, if the reference symbol was omitted, the absence of these features is known. This is not the case in PHP, where the *caller* may declare a parameter to be passed by reference.

Most existing alias analyses are designed for the semantics of Java, C and C++. They work well because they closely model those semantics. However, as PHP's semantics are only superficially similar to those of Java, C and C++, alias analyses for these static languages are largely not suitable. In particular, existing analyses:

- do not allow for modelling of variable-variables
- are not required to model dynamic effects, such as those relying on type,

- are able to fall back on a static type system [Diwan et al., 1998] in the conservative case,
- may rely on knowing the run-time structures of objects [Choi et al., 1999].

6.5 Analysis

Based on the problems with previous analyses, and our experience in compiling PHP in our previous work, our analysis has the following major features.

- We model variables as symbol-table fields, making their representation the same as for object fields and array indices. We refer to these collectively as *names*[8] from here-on-in. A *name* is a variable, object field, or array index. A s a memory location. In the canonical PHP implementation, this corresponds to a `zval`, which is discussed in Section 5.5.

  This allows us to model:

  - all tables using the same elegant abstraction,
  - variable-variables (in the symbol-table) and variable-fields in the same way as array indices,
  - references between each kind of *name*, modelling many programs that Pixy cannot,
  - the structure of arrays and objects,
  - implicit conversions of `NULL`s to arrays or objects.
- We perform type-inference simultaneously with alias-analysis, constraining each *name* to a conservative set of types. This allows us to:

  - analyse the program's interprocedural data-flow through polymorphic function calls,
  - model scalar operations, casts, and implicit type conversions,
  - track calls to magic methods,
  - prove the absence of object handlers.

[8] Xie and Aiken [2006] use the term *location* in a similar way. However, a location is traditionally a single run-time piece of memory. Our usage differs slightly in that multiple names may refer to the same run-time value, so we use a different term.

- We propagate literals and constants at the same time, in order to resolve variable-variables and conditional branches where possible.
- Finally, we use the analysis to conservatively model the definitions and uses of *names*, which is otherwise not straightforward.

The combination of these features is powerful. Only by combining alias analysis, type inference and literal analysis are we able to model types and references of a whole program. As types and aliases are so dynamic in PHP, we must combine the analyses in order to be able to track any information at all. In other words, any of these analyses on their own would be useless.

Combining them also allows us to

- model assignments to known array indices,
- determine if a parameter is passed to a method by copy or by reference,
- resolve many conditional statements,
- enable a powerful SSA form on which to base optimizations and further analyses.

Our analysis handles a large majority of PHP's features, with the exception of those listed in Section 6.5.9. As such, it is the first analysis capable of being used for a conservative analysis of real PHP programs.

6.5.1 Analysis Overview

With these features in mind, we present an overview of our analysis. Our analysis is structured as a driver analysis with five sub-analyses working as clients. The driver analysis performs a symbolic execution over the program, feeding and fetching information from the sub-analyses. The sub-analyses are

- An alias analysis which keeps track of references between *names*, and pointers from names to tables. It is told about reference operations in the program, which may create or destroy references. It tells the driver analysis all possible references of a *name* being operated on, and it is used to convert access paths to *names*.

- A type analysis which keeps track of the types of all values in the program. The driver analysis needs type information to perform implicit conversions, to know what magic methods may be implicitly called by an operation (see Section 6.2.5), and to resolve virtual-method calls. The type analysis also models operations between scalar types; for example, the sum of two integers can be an integer or a float, but the sum of two floats may only be a float.

- A literal analysis which propagates literal values for each *name*. The driver analysis can resolve conditional branches with this information. It also needs to know if values may be `NULL` in case implicit conversions to objects or arrays need to be performed. The literal analysis is told which values are assigned to which *names*.

- A constant analysis which keeps track of the values and types of user-defined symbolic constants. Although these constants are not mutable, their values may depend on the run-time control-flow of the program.

- A def-use analysis, which keeps track of what *names* are assigned to and used at each point in the program. This is not explicit in the program text, so we are require to analyse it.

The novelty of our algorithm is in the modelling of PHP features, and how the driver analysis uses this information to guide it. Bearing that in mind, we present the algorithm for the driver analysis, which is based on Pioli et al.'s [1999] algorithm:

- Our analysis performs a symbolic execution of a PHP program. We begin at the first statement in the global scope, and perform our analysis one statement at a time. At every statement, the analysis has two way communication with its sub-analyses.

- Our analysis is flow-sensitive, using a worklist algorithm to keep track of the statements in a function. It is a *conditional* analysis, meaning we attempt to resolve branch statements immediately using our literal analysis. This results in an optimistic analysis, which must complete in order for its results to be correct.

- The analysis is interprocedural, and optionally context-sensitive. Upon reaching a method invocation, we pause the current method's worklist, copy necessary information to the callee, and begin a new worklist with the entry block of the invoked method. We build our call-graph lazily, and use type information

to resolve method calls. In the case of polymorphic call-sites, we analyse all possible callees.

- When a method has been fully analysed, we copy analysis results back to the caller and continue the analysis there. If there are multiple possible receivers, we analyse each receiver in turn, merge their results, and continue the analysis in the caller. We continue the analysis using the existing worklist, which was paused before method invocation.

- After the global scope is fully analysed, the analysis is complete. We merge the results from each analysed context into a single result for each statement in the program. We then apply our optimization passes, and repeat the analysis until it converges.

Names

At each PHP statement in the program, the driver determines the lvalues and the rvalues, and converts them into an access path (see Section 6.5.2). The driver hands the access path to the alias analysis, which determines the set of *names* accessible by the lvalue or rvalue. All fields, array indices and variables are referred to by *names*, which are strings used at analysis-time to look-up results.

For example, the rvalue `$x` refers to a local variable in the scope of the function '*foo*'. The variable `$x` is therefore a field named '*x*' of a symbol-table named '*foo*'. The alias analysis will return the *name* '*ST::foo::x*' ('*ST*' for symbol-table). '*ST::foo::x*' is used to store the possible types, values and aliases of `$x` in our solution sets.

More complex lvalues and rvalues are possible. In order to access the array index `$a[$i]`, both `$a` and `$i` must be resolved as just described. The *name*`$a` will likely resolve to a piece of anonymous memory, let us call it '*anon0*'. Let us suppose that the value of `$i` is `0`. Then `$a[$i]` will resolve to '*anon0::0*'. Naturally, `$i` may be unknown, or `$a` may refer to a number of different pieces of memory. These are all discussed in the following sections.

Why a Combined Analysis?

We can see how the sub-analyses are used by the driver. The alias-analysis is used to resolve lvalues and rvalues to their *names*. During resolution, the values of *names* may

be useful, for example to resolve `$i` to `0`. The *names* are used to look up the values of the literal analysis. Type-analysis is also used to deal with implicit conversions. If `$a` is not an array in the example above, it may be converted into an array, except in special circumstances, as discussed in Section 6.2.12. In order to deal with all of PHP's edge cases, the results of the type analysis must be used throughout the rest of the analysis.

6.5.2 Alias Analysis

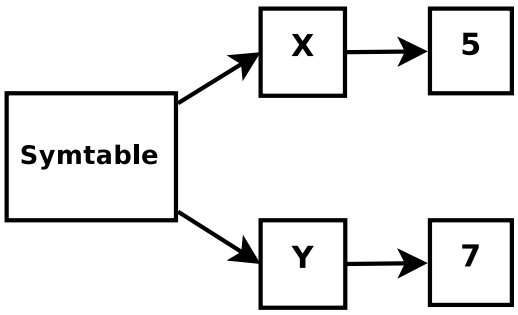
Our alias analysis is novel, particularly in how it models references between *names*. To date, no alias analysis has been capable of modelling PHP's references correctly. It is based on a *points-to-graph* [Emami et al., 1994] but it is significantly different than existing work. Primarily, we model *values* (either scalars or tables), and table fields. Using this abstraction we are able to model all of PHP's complex semantics with the same simple framework. We model references similarly to Pixy [Jovanovic et al., 2006], but allow references between any *names* in the program (fields, variables and array values). We model pointers as in traditional C or Java analyses [Emami et al., 1994]. Our points-to analysis is designed specifically to allow this, and we went through many unsuccessful iterations of this idea before settling on this powerful model.

Our alias analysis uses a points-to graph to represent the program state. Example points-to graphs are shown in Figure 6.3. It contains three types of nodes:

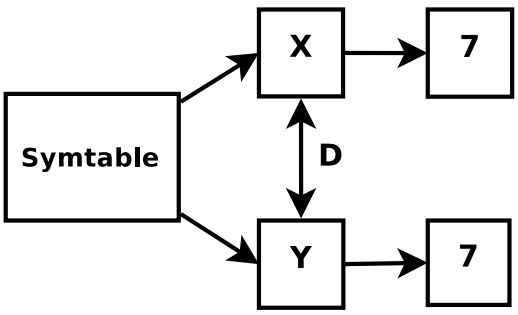
- A *storage node* represents a table. All arrays, objects and symbol-tables are represented using storage nodes. Figure 6.3 shows storage nodes marked '*Symtable*'.
- An *index node* represents a *name*, that is, a field of a table. It is used to represent variables, object fields and array indices. Each index node is a child of a single storage node. Figure 6.3 shows index nodes marked '*X*' and '*Y*'.
- A *value node* represents a scalar value. A value node belongs to a single index node, and is not shared between references. Figure 6.3 shows value nodes marked '*5*' and '*7*'.

There are three kinds of edges between nodes.

- A *field edge* is a directed edge from a storage node to an index node. The index



(a) *Points-to graph for an assignment by-copy from Listing 6.5*



(b) *Points-to graph for a reference assignment from Listing 6.6. The reference edge marked* **D** *is a definite reference edge between* `$x` *and* `$y`.

Figure 6.3: *Points-to graphs, corresponding to the PHP run-time memory models in Figure 6.2*

node is therefore a field of the storage node. Figure 6.3 shows field edges coming out of the storage nodes marked '*Symtable*'.

- A *value edge* is a directed edge from an index node to a storage or value node. If an index node has only one such edge, then the edge's target is its only possible value. Each index node must have at least one outgoing value edge.

  A value edge whose target is a value node connects a *name* to a scalar value. A value edge whose target is a storage node represents a pointer. Figure 6.3 shows values edges going into out of the values nodes marked '*5*' and '*7*'.

- A *reference edge* is a bidirectional edge between two reference nodes, indicating that two index nodes reference each other. A reference edge has a *certainty* and represents either a *possible* reference or a *definite* reference;[9] that is, two *names* may-alias, or must-alias. Figure 6.3 shows a definite reference edge marked '*D*'.

Each array or object allocated in the program is represented by some storage node in the points-to graph. These storage nodes represent dynamic memory allocation, which may be the result of a memory allocation statement using `new`, an explicit array instantiation. They may also be the result of implicit value creation, which are very common (see Section 6.2.12).

Regardless of their source, storage nodes are named based on the their allocation site and a portion of the call-string [Sharir and Pnueli, 1981]. Each storage node may be concrete (it represents a single run-time structure) or abstract (it may represent

[9] These are the terms used by Emami et al. [1994].

multiple run-time structures). The algorithm to determine concrete nodes is shown in Section 4.6.3.

A number of other program constructs require storage nodes:

- a storage node is added for the global symbol-table,
- each class requires a storage node for its static fields,
- a storage node is added for each called function's local symbol-table.

Our analysis is both field-sensitive and array-index-sensitive. In fact, fields, array indices are modelled identically to variables. The representation of variable-variables is the same as unknown array indices. When array-indices are known, they are modelled as precisely as variables.

Some index nodes (*names*) are eligible for strong updates, in which they kill their previous value or reference relationship. A strong update may be performed on an index node i if all of the following conditions are true:

- i's storage node is concrete (only represents one run-time value),
- i is not an *UNKNOWN* node (see below),
- i is the only index node referred to by an access path of an assignment (see Section 6.5.2). For example, `$x` can only refer to one index node, the node named '*x*' which is an index node of the local symbol table. However, `$x[5]` might refer to several: if `$x` points to several storage nodes, `$x[5]` refers to the set of those storage nodes' index nodes named '*5*'.

At CFG join points, we merge points-to graphs from predecessor basic blocks. We place an edge in the new points-to graph if it exists in either graph. If a reference edge exists in both graphs, then it is *definite* if it is *definite* in both graphs. In all other cases, it is *possible*.

At the end of the analysis, contexts are merged into a single solution. For example, the variable `$x` may have a different set of possible types based on the calling context. All *names* in the program include a portion of the call-string. To merge them, all calling-context is removed from the *name*, and they are merged in the same manner as at a control-flow join point.

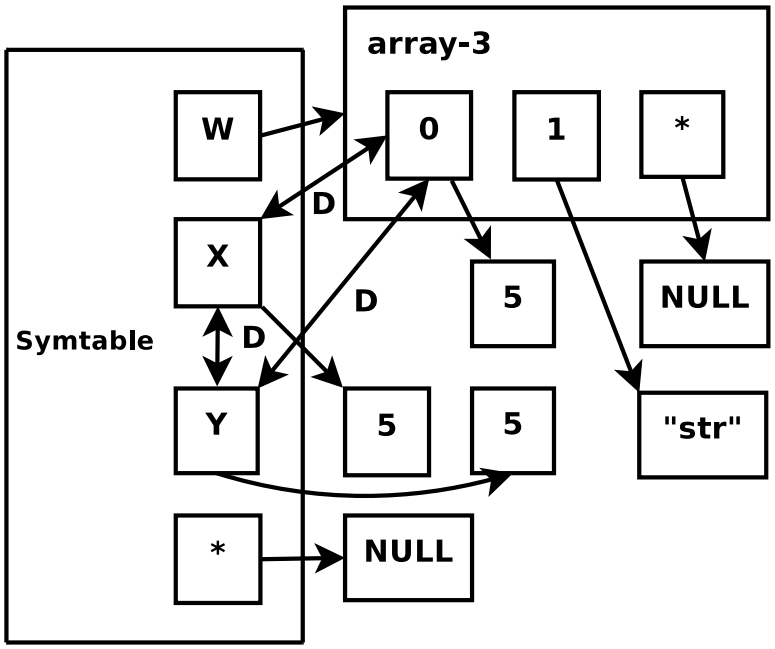


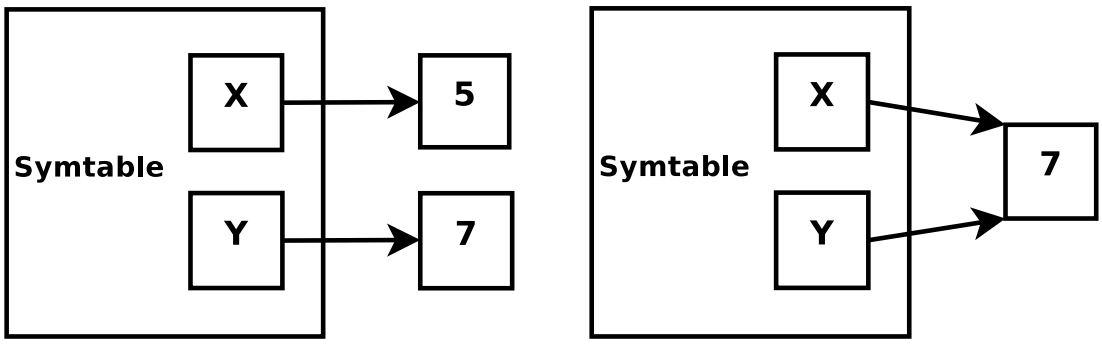
Figure 6.4: *Points-to graph.* 'UNKNOWN' *nodes are marked '*'.*

UNKNOWN node/*name*

An *UNKNOWN* node is an index node which represents all the fields of a storage node which are not explicitly named. For example, if the analysis can show that an array has two occupied indices, `0` and `1`, it will model it with three index nodes: '*0*', '*1*' and '*UNKNOWN*'. The first two represent an exact field, the last represents all other possible fields the table may have. Note that it is always necessary for a table to have an unknown node, as any field of a table may be accessed even if it has not been defined. The *UNKNOWN* node is a special index node, of which there is one per storage node. *UNKNOWN* nodes are shown in Figure 6.4, indicated by '*'.

Some analyses collapse all array indices into a single value [Xie and Aiken, 2006]. Other analyses are field sensitive [Choi et al., 1999]. In our model, array indices are represented identically to fields, in order to handle variable-variables and variable-fields. As such, the field-sensitivity we use for variables and objects extends automatically to arrays. Both Jensen et al. [2009] and Jang and Choe [2009] include *UNKNOWN* nodes in their Javascript models.

Upon creating a storage node, a new *UNKNOWN* node is added to that storage node. It is connected to a value node, whose value is `NULL`. An access to an empty table in PHP evaluates to `NULL`, which this models.



(a) *Assignment by copy (see Listing 6.5)* **(b)** *Assignment by reference (see Listing 6.6)*

Figure 6.5: *PHP assignment memory representation*

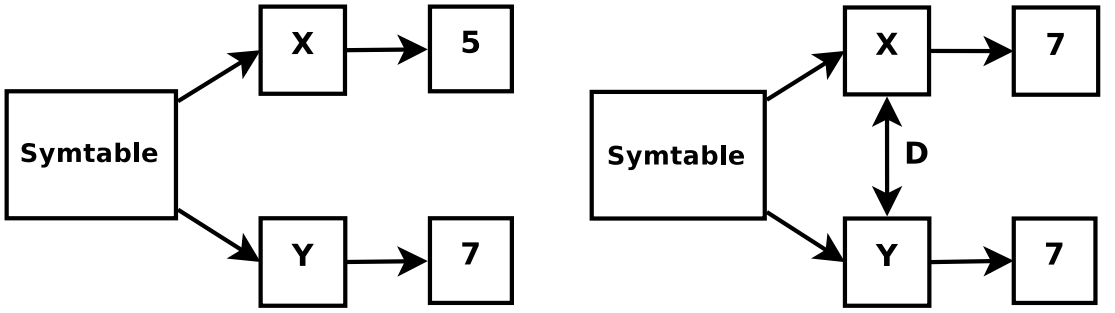
An *UNKNOWN* node represents the values of index nodes which are *not* present in the storage node. When an field whose *name* is not statically known is set, the *UNKNOWN* value for the storage node is set, as is the value of every other index node in the storage node. This models that every single field in the storage node might have been set.[10] Reading from a statically unknown *name* likewise reads from all possible fields of a storage node, not just from *UNKNOWN*. If a *name* is read which is not present in the points-to graph, the value from the *UNKNOWN* node of the appropriate storage node is used instead.

UNKNOWN nodes are also used to merge two points-to graphs p and p'. If a field edge $s \rightarrow i$ exists in p which is missing from p', then the value of $s \rightarrow UNKNOWN$ from p' is used for merging. The i node has been assigned in p, but may or may not have been assigned in p', so the value in p' must come from the *UNKNOWN* node of s. The only exception to these rules is that if s does not exist in p', s's index nodes are copied directly from p, as there is no *UNKNOWN* node of s in p' to merge them with. These merges are common, even for variables and fields, as *names* are often initialized in loops, and different storage nodes are built in different paths due to context-sensitivity.

Comparison to PHP Memory Model

The difference between the PHP run-time memory model and our points-to graphs is demonstrated in Figure 6.6, corresponding to Listings 6.5 and 6.6. The PHP run-time

[10] As an extension to this scheme, if we could constrain the field name using string-range analysis [Wassermann and Su, 2007], the analysis would be more precise.



(a) *Points-to graph for an assignment by-copy from Listing 6.5*

(b) *Points-to graph for a reference assignment from Listing 6.6. The reference edge marked* **D** *is a definite reference edge between* `$x` *and* `$y`.

Figure 6.6: *Points-to graphs, corresponding to the PHP run-time memory models in Figure 6.5*

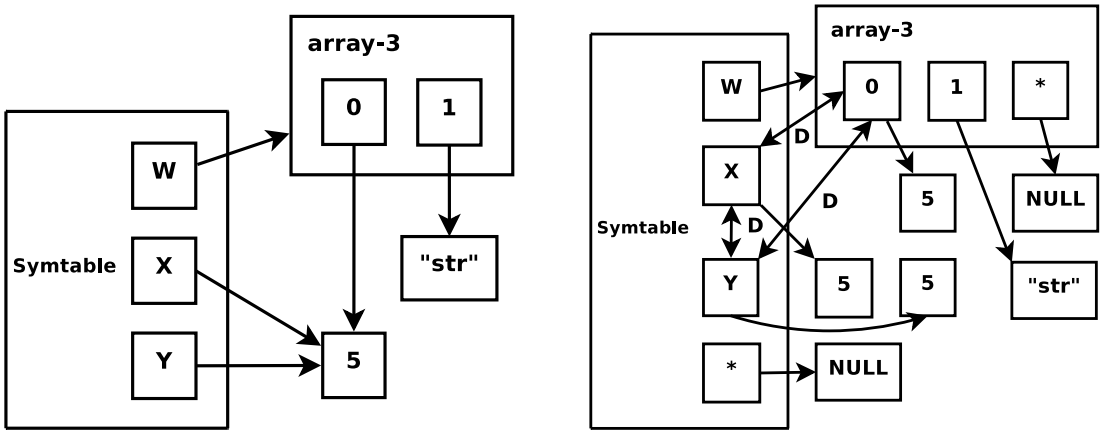
```
1 $x =  5;
2 $y =& $x;
3 $w = array ();
4 $w[0] =& $x;
5 $w[1] = "str";
```

Listing 6.10: *A short program with reference assignments and an array*

memory model for these listings is shown in Figure 6.5. For a simple assignment-by-copy, the compile-time and run-time models are largely the same.

However, the difference is more visible in an assignment-by-reference. We designed our analysis to model both may-aliases and must-aliases, so we are not able to simply use multiple value edges to the same value to model references, as in the PHP run-time memory model. Instead, we are able to model this by using reference edges between aliased index nodes. If we modelled the run-time behaviour by simply sharing value and storage nodes, we would be unable to model must-reference behaviour, resulting in a significant loss of precision. We also use one value node per index node, rather than sharing value nodes between references. If we shared value nodes, then each may-definition (an assignment via a may-reference) would need to be a weak update. Instead, this allows a strong update to be be performed on the index node being defined, and only its may-references are weak-updated.

For a larger example, Figure 6.7b shows the points-to graph for Listing 6.10. The corresponding run-time memory layout is shown in Figure 6.7a. At run-time, the symtable has three local variables, `$w`, `$x` and `$y`. The variable `$w` points to an array. The variables `$x` and `$y` alias each other and also alias the 0'th element of `$w`'s array. The array's 1'th value is also shown. We can see again that references between index



(a) *Run-time memory representation*

(b) *Points-to graph. For simplicity, index nodes are shown within their storage node, instead of using field edges. Edges marked* D *are definite reference edges.*

Figure 6.7: *Memory layouts for Listing 6.10*

nodes are shown not by sharing the same value nodes, but rather by the edges between the aliased index nodes.

Access Paths

As the abstraction used for all tables in the program is the same, we are able to map all lvalues[11] and rvalues[11] to a traversal of the points-to graph. This mapping is referred to as an *access path* [Larus and Hilfinger, 1988].

An access path consists of a *storage part* and an *index part*, representing the table being indexed, and its key. Consider a simple assignment `$l = $a[$i]`. The lvalue is modelled as:

$$\$l \Longmapsto ST \rightarrow l;$$

that is, `$l` maps to the local symbol-table indexed by the string '*l*'. Both the storage and index parts can themselves be access paths, required for array dereferences:

$$\$a[\$i] \Longmapsto (ST \rightarrow a) \rightarrow (ST \rightarrow i)$$

[11]As described in Section 3.1, we use *lvalues* to refer to *names* which may be written to, and *rvalues* to refer to *names* which may only be read. However, because of PHP's ability to alter values that are being read, this isn't a strict definition.

In this case, both the array and its index must be first be fetched from the symbol-table. We would expect in general that `$a` will find a storage node corresponding to an array, and the `$i` will find a value node representing an integer or a string. Variable fields are modelled the same way.

Variable-variables are modelled as

$$\$\$x \Longmapsto ST \rightarrow (ST \rightarrow x)$$

while simpler array assignments are represented as

$$\$a[0] \Longmapsto (ST \rightarrow a) \rightarrow 0$$

An access path evaluates to a list of index nodes. If the access path represents an lvalue, the value(s) of the index nodes are being set. If the path represents an rvalue, the value(s) of the index nodes will be copied or referenced. The algorithm for converting an access path to set of index nodes is straightforward:

- Each access path where both the storage part s and the index part i are named is evaluated. The access path will resolve to a storage node named s with a field named i.
- If the resolved access path p' is part of another path p, then the values of the index nodes represented by p' are used to resolve p.
- An access path with an unknown index will evaluate to every possible index node of a storage node, including the *UNKNOWN* index node.

6.5.3 Literal and Type Analysis

We track information about the literals[12] and types of each *name* in the analysis. This means that *names*, such as variables or fields, may have more than one possible type, as is to be expected in a dynamically typed language.

We model literal values using a semi-lattice, as shown in Figure 6.8. At a CFG join

[12]We remind the reader that *constants* have a different meaning in PHP, which are modelled separately.

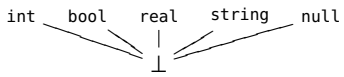


Figure 6.8: *Literal propagation semi-lattice*

point, two unequal values merge to ⊥.[13]

The reason we do not use a lattice is that uninitialized values in PHP evaluate to `NULL`. At a CFG join point, a *name* with a value (v_1) may meet an uninitialized value (v_2). If we used a lattice, the meet of these two would be v_1 instead of ⊥ (assuming that v_1 is not `NULL`. As such, a lattice topped with ⊤ would not be a correct model. We note that Pixy [Jovanovic et al., 2006] used a lattice, which may lead to incorrectly propagated constants.

The value of an uninitialized *name* is not guaranteed to be `NULL`, but instead takes its value from the *UNKNOWN* value of the appropriate storage node. If the table does not exist, then the uninitialized value is known to be `NULL`. This is important when performing joins at control-flow merges.

We use a set to model the types of each *name*. *Names* with known values have the type of their value. Otherwise, value nodes are permitted to have any scalar type. Storage nodes are either '*array*'s for arrays and symbol-table, or the single concrete type of an object. Index nodes use the set of the types of all values to which they point. At CFG join points, the sets of merged using set union.

6.5.4 Constant Analysis

For each constant defined in the program, we model its literal value and type, if known. If a constant has not been defined, it will evaluate to its symbolic string (e.g. *MYCONST* would have the value `"MYCONST"`). Constants may only have scalar values, so unknown constants have bounded types. Our analysis can call into the PHP run-time, as discussed in Chapter 5, so we have automatic access to constants defined in PHP's standard library, which augments the results of our analysis.

[13] We use ⊤ for uninitialized, and ⊥ for unknown.

6.5.5 Definitions, Uses and SSA Form

PHP's syntax provides few clues as to the variables which are defined or used in a simple statement. Although aliases exist in more traditional languages, PHP's feature set means we cannot make a conservative estimate of the def-use chains [Muchnick, 1997] in a PHP program. Principally, it is not statically known whether values are passed as arguments to methods by-copy or by-value. In addition, features such as variable-variables, functions which affect the symbol-table of their callers, and object handlers mean that it is impossible to use a purely syntax-based approach.

While performing our analysis, we record in each statement all variables which are defined and used in that statement. That is, in order to correctly model simple definitions and uses, we must run a def-use analysis in parallel with our type- and alias-analysis. By contrast, a conservative def-use chain can be built for C++ using type-based alias analysis [Diwan et al., 1998]. C++ does not allow a program to modify symbol-tables, and even C++'s most hidden effects are statically known. This means that after an alias analysis is performed, it is trivial to build a def-use chain.

We use our def-use analysis to create an SSA form, which would otherwise be impossible to correctly build. The SSA form, based on *Hashed SSA* (HSSA) [Chow et al., 1996], creates a platform for further analysis and optimization, which can be built without having to be integrated into the alias analysis. This is discussed in more detail in Section 4.6.4. Using this, we build a powerful aggressive dead-code elimination pass [Torczon and Cooper, 2007].[14] We present results for this analysis in Section 6.6.

For simplicity, we have up until this point said that the def-use chains and SSA form were built using variables. In fact, we again use the *name* abstraction, allowing us to model fields and array-offsets in exactly the same manner as variables. As such, we can remove not only assignments to dead variables, but dead stores as well.

Our dead-code elimination is also able to remove reference assignments in addition to standard scalar variables. The ability to delete reference assignments comes from our def-use model. Each *name* may, in each statement, be defined, may-defined[15] or used, both *by reference or by value* (that is, there are six possible combinations. *Value* uses and definitions relate to scalar values, while *reference* uses and definitions model def-use chains over a program's reference assignments.

[14] By dead-code elimination, we mean useless code, not unreachable code (although our analysis never analyses statically unreachable code).
[15] The same could be said of uses, but it isn't useful to do so.

For an assignment by-copy, `$x = $y`:

1. `$x`'s value is defined.
2. `$x`'s reference is used (by the assignment to `$x`).
3. for each alias `$x'` of `$x`, `$x'`'s value is defined. If the alias is *possible*, `$x'`'s value is may-defined instead of defined. In addition, `$x'`'s reference is used.
4. `$y`'s value is used.
5. `$y`'s reference is used.

For an assignment by-reference, `$x =&$y`:

1. `$x`' value is defined.
2. `$x`'s reference is defined (it is not used—`$x` does not maintain its previous reference relationships).
3. `$y`'s value is used.
4. `$y`'s reference is used.

We note that the other *names* referenced by `$y` do not need to be used in an assignment by-reference. Rather, their use can be determined by traversing the reference def-use chain starting at `$y`.

Finally, we note that we model def-use chains over `foreach`-loop iterators, discussed in Section 4.2.4. This allows our dead-code elimination pass to remove `foreach`-loops which are unused.

6.5.6 Modelling Library Functions and Operators

PHP has a very large number of built-in and library functions, which are written in C, and therefore not analysable. The manual documents over 5000 of these function. As we are required to know the possible set of types a *name* (variable, field or array-offset) may have at all times, we are required to model these functions. We model three aspects:

parameters: Knowing whether a parameter must be passed by reference simplifies analysis and generated code. We retrieve this information automatically using our link to the PHP run-time, as described in Section 4.5.3.

Some functions (typically string or array functions) alter the values of their reference parameters. Modelling these functions precisely can take a some time for the author of the analysis, so we typically model them as simply as possible, which may lead to slightly more conservative results than are possible.

A parameter which expects a string may be passed an object with a `__toString` magic method (see Section 6.2.5) instead, which will be called when evaluating the string value. As such, we model parameters to indicate if they expect a string value, in which case we model calling the `__toString` method. There might also be other magic methods which could be handled in this way, but we have only modelled `__toString`.

return values: For most functions it is sufficient to model the type of the return value, usually a small set of scalars. A large number of functions return the value *false* in addition to their intended type, to denote an error.

A small minority of functions return some array structure, such as an array of strings. In those cases our model returns an array with the appropriate structure.

purity: There are a large number of functions which are pure (side-effect free). If we know the literal value of all parameters, we may execute this function at compile-time, and return the correct value. Again, this uses our link to the PHP run-time. As we actually call pure functions at compile-time, we may cause the compiler to spend arbitrarily long amounts of time on analysis. We use a short timeout to avoid these problems, but in practice the time taken to execute pure functions is negligible.

We also model operators in the same manner: executing them if their operands are known, or modelling their result if it is not. In most cases, the results of simple operands are not simple: the sum of two integers may be a real, the product may be an integer, real or the value false. The semantics of these operations were gleaned from reading the source of the canonical implementation—they are not otherwise documented—but their documentation is outside the scope of this dissertation.

6.5.7 Termination

We argue informally that our algorithm will terminate. The semi-lattice is clearly bounded in depth, so literals and constants will converge quickly. The set of types is bounded to the number of types in the program. As we require all classes to be available at analysis time, this provides a bound on the size of the sets.

Our alias analysis may add at most one storage node per context in the analysis. The number of index nodes a storage node may have is bounded by the number field/variable/array assignments in the program. An assignment to an unknown index node will use *UNKNOWN*. An assignment to an index node whose *name* is derived by the literal analysis will converge as the literal analysis does.

In the presence of recursion, we alter our call-string to indicate recursion. The call-string omits any of the function calls after the function which is the head of the recursive loop in the call graph. For example, the call-string '*foo:bar:foo:bar:foo:bar*', which is analysing the mutually recursive functions `foo` and `bar`, would be shortened to '*foo:bar:RECURSION*'. As all of the recursive functions use the same analyses, the recursive analysis will converge if the other analyses do.[16]

6.5.8 Deployment-time Analysis

Programming languages may be implemented using interpreters, compilers or just-in-time compilers. In some cases, the design of the language affects how practical it is to implement in a compiler or interpreter. Many scripting languages are designed in such a way that a lot of program source code is not available at '*compile-time*'.

In languages designed for ahead-of-time compilation, such as C and C++, all of a program's code is typically available during compile-time. However, a portion of code may not be available at compile-time, such as when a program allows plug-ins, implemented with dynamic loading. Even in the presence of dynamic loading, large portions of the program are inaccessible to dynamically-loaded code, and these portions are well-defined using statically typed interfaces.

PHP is designed for interpretation in a web server rather than ahead-of-time compilation, and this affects the design of PHP programs. PHP programs normally require configuration during deployment on a web server. Typical settings include the user-

[16] We note that `phc`'s support for recursion is buggy, due to implementation issues.

name and password of the database and the domain name on which the application is running. As PHP is normally interpreted, many programs simply `include` a source file which contains application settings, rather than parsing a separate configuration file when the program starts. Another common pattern is to implement localisation and plugins by conditionally including source files at run-time. It is not possible to tell whether these files contain only simple strings, or whether they dramatically alter the state of the program.

As a result, it is not practical to analyse PHP programs at distribution-time or install-time, as is commonly the case for more static languages. Instead, we introduce the notion of *deployment-time* as the appropriate time to perform program analysis. *Deployment-time* is the first time when all of a program's code will be available.

As PHP programs are run on the server, a program's end user does not have access to them. Program's are deployed by administrators, and plugins, language packs and configuration files are all changed infrequently. As our analysis requires whole-program analysis, `phc` relies on a model where it is run *after* all of these source files are installed. In the case of a program update or reconfiguration, or the installation of extra plugins or language files by the server administrator, it is not a large burden to recompile the program.

Program analysis is not always used for compilation. In these cases, it may be possible to make non-conservative approximations of files unavailable at analysis-time. In these cases, deployment time analysis is not necessary.

6.5.9 Future Work

While we have modelled nearly the entire PHP language, our model is incapable of expressing some parts of PHP. The *eval* statement evaluates source code at run-time in the current scope. If we do not know the value of the string beings evaluated, we are unable to know anything about its effects, and our analysis is forced to return unknown for all *names* in the program. In practice, we stop the analysis upon discovering an *eval* statement. Other analyses for scripting languages have reported the same problem [Jensen et al., 2009, Wassermann and Su, 2007]. Furr et al. [2009] have an interesting partial solution to this problem, discussed in Section 3.6.1.

We have not modelled magic methods other than `__toString`, but we have instead chosen to prove their absence. The analysis stops upon seeing a class which defines a

magic method. We have also chosen to prove the absence of object handlers, instead of modelling them. An object handler is a property of an object, not a class, and so our class based type-inference is slightly too weak to model them. It would not require a large addition, however.

Values which are reachable from the `_SESSION` variable are used for persistence between PHP program invocations. This means that the values fetched from the `_SESSION` variable may have values or types which are not analysable. We optimistically model it as an array of scalars. It would be straightforward to handle the `_SESSION` variable correctly by iterating over the other PHP scripts in the application which use the `$_SESSION` variable, or with user annotations. The former is similar to class analyses for C++ [Dean et al., 1995].

We also do not model incorrectly-called functions and methods. The correct behaviour is to issue a warning and return `NULL`, but we assume that the function is called correctly.

We do not model error handling or exceptions. Both of these are highly dynamic in PHP, and this is the most severe limitation of our analysis.

We do not support dynamic class and method declarations, but these were not intentionally used by programs we have seen (they appeared from **`include`** statements within functions).

We note some areas in which our analysis could be improved. We can often say that a variable must be initialized (if it has a non-`NULL` value), but we cannot say that it is not initialized. Doing so would provide better opportunities for optimization.

6.6 Experimental Evaluation

In this section, we provide an experimental evaluation of our research, analysing a number of benchmarks, and comparing different versions of our algorithm. We implemented our analysis in `phc`, our ahead-of-time compiler for PHP. The `phc` compiler is open-source, and the analysis algorithm is publicly available.[17]

We analyse five publicly available PHP programs, including the RUBBoS and RUBiS benchmark suites [Amza et al., 2002], the Zend benchmark[18] used to benchmark

[17] from `http://www.phpcompiler.org`
[18] We exclude recursive benchmarks from the Zend benchmarks.

	Scripts			Lines of code			Statements	
Name	Analysed	User-facing	Package	User-facing	After inclusion	Package	Static	Analysed
RUBBoS	16	18	19	1597	2900	1597	3423	1001
RUBiS	19	20	21	1747	3868	2095	4435	1837
Eve	3	4	8	215	440	907	1473	306
Zend	12	14	14	421	421	421	1398	890
SQLiteAdmin	9	10	16	2791	12005	2915	2583	1212

Table 6.1: *Characteristics of analysed benchmarks*

PHP's canonical implementation [The PHP Group, 2007], and several programs which were analysed in previous research (Eve Activity Tracker 1.0 and phpSQLiteAdmin 0.2). We note that our analysis is performed at *deployment-time*, as described in Section 6.5.8.

In Figure 6.1 we list statistics from the programs analysed. Each program comprises a number of user-facing PHP scripts. Each user-facing script requires the source of many backing scripts. As we analyse each script separately, we perform a pre-pass to include all of the backing scripts code automatically.

The *Scripts* column lists the number of user-facing scripts, the number of those scripts we analysed[19] (A), and the total number of scripts in the package. The *Lines of code* column lists the number of source lines of code (SLOC) in analysed user-facing scripts, the sum of the SLOC of those scripts after the inclusion pre-pass, and the sum of the SLOC of each file in the package. The *Statements* column shows the number of static statements in the program, and the number of statements in the program which were traversed during analysis. Unreachable code, such as functions which are never called, are not traversed during analysis, meaning the number of statements analysed is generally lower than the static number, even taking flow- and context- sensitivity into account. Many unreachable functions are present in the program due to the inclusion pre-pass, and the structure of PHP programs. All further figures are aggregated over each analysed script in a test package.

We analyse each program four times, varying the context- and flow-sensitivity. Our results show the difference between a context-sensitive and context-insensitive analysis,

[19]We skipped some scripts due to minor bugs.

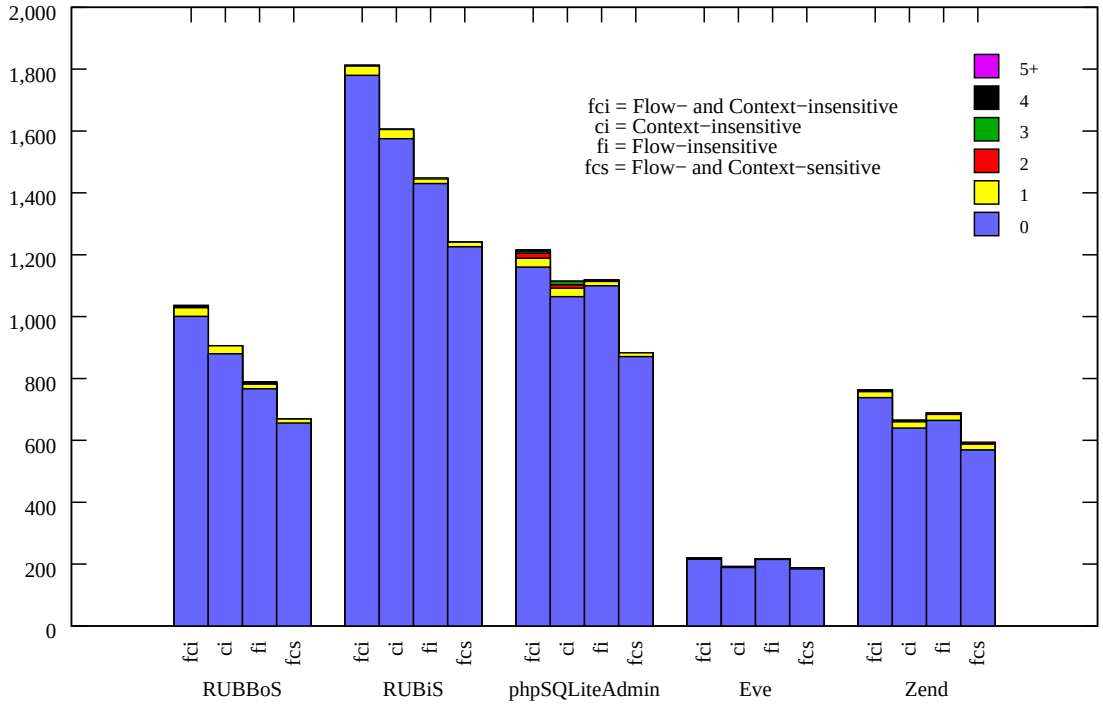


Figure 6.9: *Static approximation of the peak number of references per variable in the analysed benchmarks, parametrised by flow- and context-sensitivity. The key shows the number of references each variable in the program has. A value of* `0` *indicates a variable is not referenced by any other variable.*

and the difference between flow-sensitive and flow-insensitive analysis. Our context-sensitive version uses an unbounded call-string, our context-insensitive version uses a call-string length of one (that is, only a single statement ID is used in the name of the storage node created). Our flow-sensitive version is described in the previous section; our flow-insensitive version is simply the flow-sensitive version, except we always perform weak updates instead of strong updates. The *# statements* column in Figure 6.1 lists the static number of three-address-code statements in the program, and the number of statements analysed in the context- and flow-sensitive version of our analysis. We do not present the number of statements analysed in the flow-insensitive version, because it is only an simulation of a true flow-insensitive version.

We ran our analysis, followed by a number of optimizations including constant propagation and folding, dead-code elimination, and removal of calls to pure or empty functions. We modelled 220 simple built-in PHP functions as described in Section 6.5.6. We also manually modelled 56 more complex functions, for example pushing and popping from arrays, array merges, callbacks, and functions which return arrays.

Figure 6.9 presents the variables in a program, along with the number of times they are

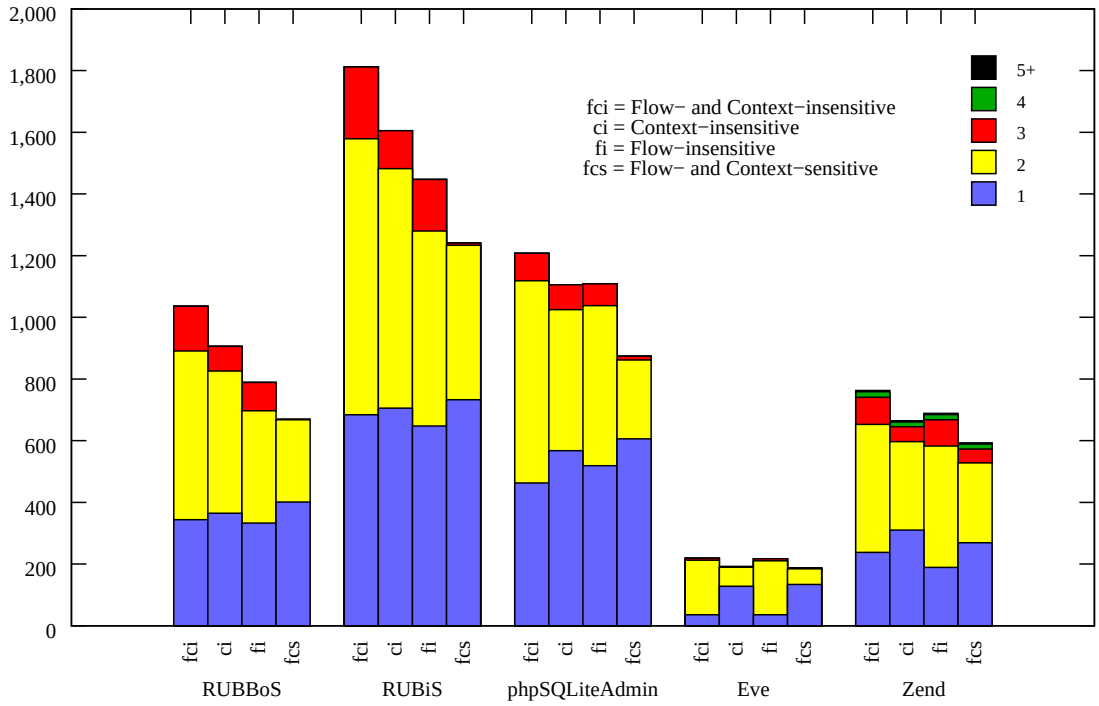


Figure 6.10: *Static approximation of the peak number of types per variable in the analysed benchmarks, parametrised by flow- and context-sensitivity. The key shows the number of types each variable in the program has. A value of* `1` *indicates the variable has only one run-time type.*

aliased. All scalar optimizations rely on variables being unaliased. Unaliased variables may be put into registers, or optimized using the *scalar replacement of aggregates* optimization [Muchnick, 1997]. The number of variables is lower in the more precise versions of the program, principally due to dead-code elimination. For each variable in a function, we count the peak number of aliases it has over its lifetime.

Our analysis is able to prove that the vast majority of variables in the program are unaliased. This shows that PHP programs have the capacity to be compiled to a much more efficient representation than their current form. No existing analysis for PHP has been able to prove anything of this magnitude.

We show too that the number of unaliased variables is largely invariant in the flow- and context-sensitivity. However, we show that flow- and context-sensitivity generally decrease the number of variables in a program with context-sensitivity being the more significant factor. More precise analyses lead to fewer program variables because dead code elimination can be much more effective. Context-sensitivity is more significant because more accurate context-sensitive analysis leads to fewer abstract storage nodes, allowing for better constant propagation.

Figure 6.10 shows a distribution of the number of types of variables in a program.

Statically knowing that a variable has only a single type allows bug-finding tools to eliminate false positives. A low number of static types also has significant benefits for IDEs and code browsers, which often auto-complete called methods and functions based on a variable's static type. More importantly, knowing the single static type of a variable allows compilers to generate better code, in particular eliminating type checks ahead-of-time. Complicated run-time type feedback algorithms have been built [Gal et al., 2009], which may not be necessary in the presence of effective type inference.

For each variable in a function, we count the number of types it has over its lifetime. This is performed after constant propagation, and so does not include statements which can be optimized to take a constant. Our results show that we can prove that 60% of variables in PHP programs have a single type, and 90% of variables have two possible types or fewer, using flow- and context-sensitive analysis. This is an important result, as it shows that a very significant number of type checks can be omitted from generated code.

In our experience, many variable which have greater than one type have a possible `NULL` value. This is particularly prevalent in loops, where variables are often uninitialized on the first loop execution, and have static types for the rest of the loop's execution. We speculate that peeling the first loop iteration would be beneficial in an optimizing compiler, which we intend to investigate in future work. In addition, arithmetic operations and calls to built-in functions rarely return a single type, with the former usually resulting in the set (*real*, *int*), and the latter possibly returning **`false`** as well as its intended value in many cases.

Figure 6.11 shows the number of branch statements in our program, and the number that we are able to resolve to a known direction. The peak in each column shows the sum of the number of branches at the start of the program and those created by small optimizations which we have not discussed.[20] *Remaining Branches* shows the number of branches which remain after the analysis has converged and all optimizations have been performed. The removed branches are split into those removed by dead-code elimination (*Branches removed by DCE*), and those where the branch direction is known (*Branches replaced with constants*). We can see that flow-sensitive analysis gives better results, significantly better in some cases.

We note that during generation of three-address-code (i.e. before analysis) `phc` creates many redundant branches, and many of these can be resolved with analysis. In particular, redundant branches comes from the `param_is_ref` construct (see Section 4.2.4), the

[20]These optimizations are the reason that the *Zend* results are not flat.

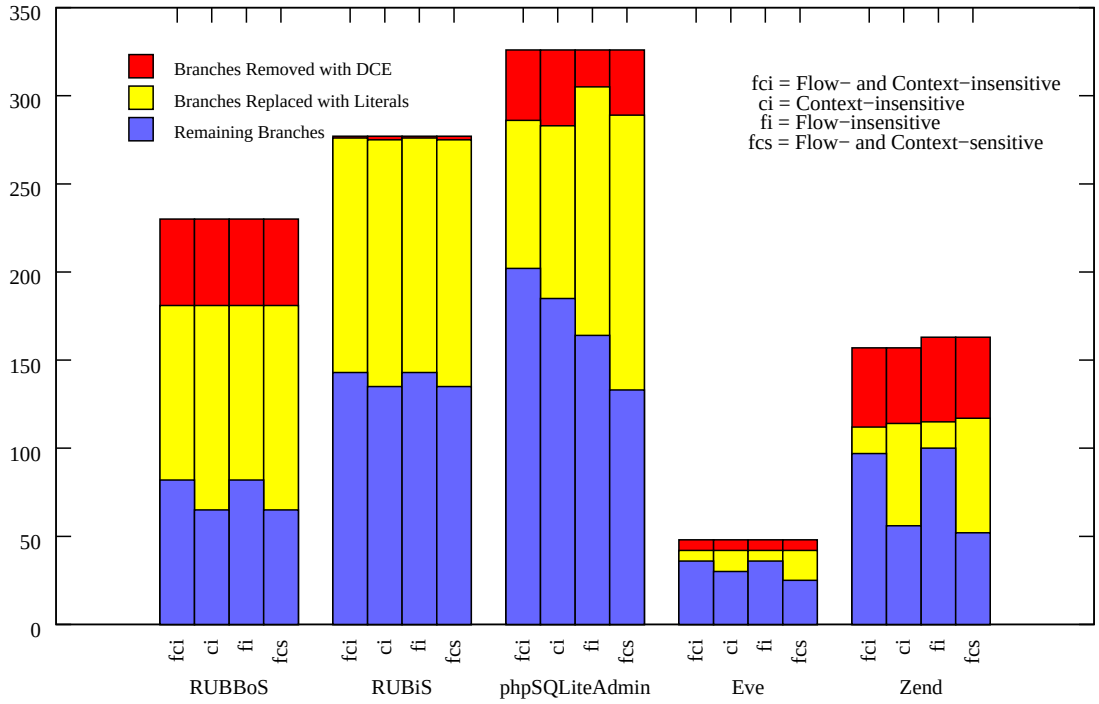


Figure 6.11: *Static branches removed in the analysed benchmarks, parametrised by flow- and context-sensitivity.*

conversion of loops into a canonical loop type (see Section 4.2.4) and the addition of type checks to handle edge cases which rarely come up in practice. The `param_is_ref` construct provides a conditional branch for each parameter of every method call, many of which are resolvable statically. However, virtual methods may have different signatures in different classes, so the construct is not resolvable in all cases, and requires type inference to be resolved.

Finally, we present the number of three-address-code statements eliminated by dead-code elimination in Figure 6.12. Our intermediate representation creates a very large number of opportunities for dead code. Constants and literals are moved into variables during the creation of three-address-code, and `param_is_ref` statements are added. We see that a great number of these statements are removed through analysis, in one case over 70%. We note that branches are not included in the number of statements, and that methods which are never called are never counted.

A clear result of our analysis is the effectiveness of static analysis for scripting languages. Though we have seen that the analysis is typically much more difficult to perform than for more static languages, the results are excellent. Our analysis is capable of identifying a very large number of statically typed variables, and a very low number of aliased variables. Our results show that many program analysis tools are

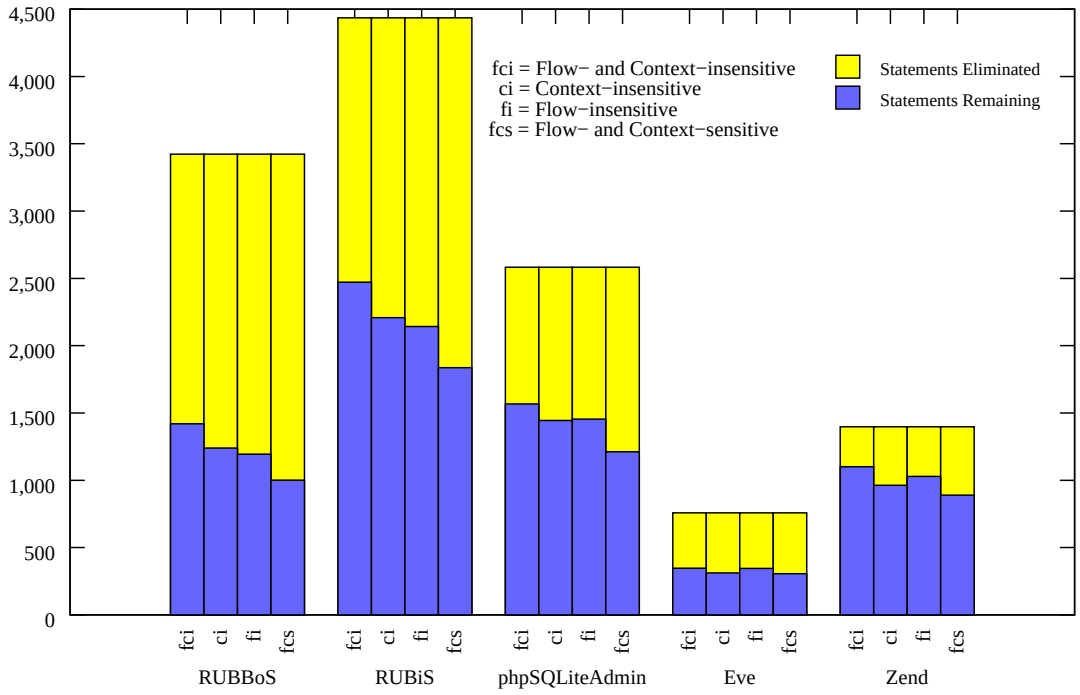


Figure 6.12: *Static dead code eliminated in the analysed benchmarks, parametrised by flow- and context-sensitivity.*

practical for scripting languages, including tools for automatic bug finding, program verification, refactoring and code browsing and editing. In addition, the optimization opportunities are clearly excellent, with large quantities of code removed, a majority of variables which are unaliased, and a large number of variables which may only be a single type over their life time.

We have seen many claims that just-in-time (JIT) compilers are the only way forward for compilers for dynamic scripting languages. While open problems remain, such as the best way to deal with `eval` statements, we believe that these preliminary results indicate that static analysis may be viable for dynamic languages.

6.7 Related Work

Our analysis is superficially similar to that of Jensen et al. [2009]. They model points-to information, type information and constant values using a lattice model. Their analysis is for the Javascript language, which is similar to PHP in that it is a dynamically typed language which uses tables as its major data structure.

Although Javascript and PHP are similar languages, their properties significantly affect

what needs to be modelled in a program analysis. While Jensen et al.'s analysis uses points-to, type and constant information, as does ours, the analyses are very different.

The principal component of our analysis is an alias analysis, designed specifically for PHP's references. PHP references make the alias analysis significantly different to an analysis for another language. In particular, we need to model to model reference edges in our points-to graph, while they only need model points-to edges. By contrast, Jensen et al.'s work models a simple points-to relationship, which is so straightforward that Jensen et al. do not discuss it in any detail. Likewise, Jensen et al. focus on their own areas of import: a dynamic class hierarchy based on Javascript's protocol based inheritance.

They also differ in their approach. Our analysis is based on that of Pioli et al. [1999], while theirs is based on Andersen's approach. The two approaches are quite different, as described in Section 3.2.2. Deriving simple types using a complex alias analysis (our research) is quite different from deriving simple pointer relations using a complex type analysis (their work) Their analysis is both context-sensitive and flow-sensitive, like ours. One important difference is that our analysis is conditional. In practical terms, this allows our analysis be more precise, while their analysis is more scalable. Our analysis is based on a set of client analyses, while theirs combines all the analyses into a complex lattice. One of their major contributions is a description of such a lattice. By contrast, our analyses seem easier to compose, which will easily allow us add string-range analysis [Wassermann and Su, 2007], as we discuss in Section 7.1.3.

Overall, the two analyses are similar in that they are both very advanced analyses for their respective scripting language. They recognise, as do we, that dynamically typed languages need to combine their alias-, type- and constant analyses. However, although similar, they are ultimately solving a different problem, with a different solution, and their novel contribution is significantly different to ours.

The same can be said about Self's type inferences [Agesen, 1995, Palsberg and Schwartzbach, 1991], as well as the scripting language analysis based on them [Cannon, 2005, Dufour, 2006, Furr et al., 2009, Salib, 2004]. In addition, the CPA algorithm describes an efficient approach towards context-sensitivity based on cartesian products of sets of types. Our context-sensitivity is more precise, but likely less scalable than CPA. CPA does not extend to handling range analysis well [Agesen, 1995]; this composes well with our analysis, and is the next logical step to more precise types for numeric programs.

Jang and Choe [2009] present a points-to analysis for Javascript. Its primary contribution is that it uses the same mechanism for array and property accesses. They do not perform type inference, and their pointer analysis is not as advanced as Jensen et al.'s work, in particular being both flow- and context-insensitive.

Other static analyses for PHP are discussed in Section 6.3. In general, a major contribution of our analysis over them is that we properly handle much more of PHP's semantics than they. In particular, Pixy [Jovanovic, 2007, Jovanovic et al., 2006,] models PHP references in a way which does not handle fields or array indices.

Control Flow Analysis (CFA) [Shivers, 1991] is a technique aimed at modelling the control-flow of higher-order languages (HOLs) such as Scheme. CFA is important because it shows how to precisely model first-class functions. Without the ability to model control-flow, HOL cannot take advantage of the wealth of dataflow optimizations which exist for imperative languages such as C and C++.

Scheme is a dynamic language, and it seems that such techniques developed for Scheme should translate to analysis of dynamic scripting languages such as PHP. However, Shivers [1991, Section 1.3.3] demonstrates why there is no natural translation. Although C allows first-class functions like Scheme, "there are differences in frequency of use" of higher-order features in these languages. In that regard, many scripting languages more closely resemble imperative languages than higher-order ones. In particular, first-class functions exist in PHP, but are seldom used in practice.

The closest algorithm to ours in that of Pioli et al. [1999], on which ours is based. We use their technique for combining alias analysis with conditional constant propagation. We further extend this to model types, which makes the conditional analysis more powerful, by enabling type queries to be resolved conditionally. However, PHP bears little resemblance to C++, and our alias algorithm is designed specifically for PHP's run-time behaviour. As such, only the driver analysis resembles Pioli et al.'s, and the rest of the algorithm is considerably different.

6.8 Conclusion

The dynamic features of scripting languages make static analysis very challenging, particularly for PHP which has no documented semantics outside the source code of its reference implementation. However, we have extensively documented PHP's run-time behaviour, how it affect program analysis, and in particular its difficult-to-analyse

features, for the first time.

We have developed a static analysis model for PHP that can deal with dynamic language features such duck-typing, dynamic typing with implicit type conversions, overloading of simple operations, implicit object and array creation and run-time aliasing. The main focus of our work is alias analysis, but we show how type inference and constant propagation must be used to perform the analysis effectively. We also show how SSA form cannot be used without the presence of a powerful alias analysis.

Our analysis has been implemented in the phc ahead-of-time compiler for PHP, and used to analyse a number of real PHP programs totalling 19684 lines of source code. Our analysis is able to determine that almost all variables in our benchmarked programs are unaliased. We are also able to statically determine the dynamic type of 60% of variables in our test programs. Finally, we have provided comparisons of context- and flow-sensitive and -insensitive variants of our algorithm and find that both context- and flow-sensitivity are valuable in increasing the accuracy of the analysis.

Closing Thoughts

This dissertation combines the fields of program optimization and dynamic scripting languages. This chapter describes the contributions of this dissertation.

During the development of phc and the writing of this dissertation, it became clear that current approaches to scripting language design and implementation could and should be improved. In this chapter, we look for better ways to design and implement scripting languages. In particular, there is a large mismatch between the design of scripting languages and the information required to effectively analyse programs and generate efficient code from compilers. This chapter describes the search for a better way.

7.1 The Future of PHP Research

7.1.1 High-level Optimizations

I began my PhD with a view to performing very high-level optimizations. Users of low-level languages such as C and C++ waste significant amounts of time manually optimizing their program. It would generally be preferable if the compiler could optimize all facets of a program automatically. However, the compiler is generally unable to put together complicated low-level patterns to establish the higher-level operation being performed, hindering optimization. By exposing low-level details such as pointers, the compiler is actually crippled by C/C++'s language features. It seems obvious that the extra information encoded in high-level constructs would allow greater levels of optimization in high-level languages.

An interesting high-level optimization for scripting languages is the optimization of tables in dynamic scripting languages. Tables are used extensively in PHP and other scripting languages, where they are used for all simple data structuring needs. As such, tables are used as vectors, as stacks, as structs with a known set of fields, as sparse arrays of integers, or as sets. Even when used as intended, a table's keys could record regular string data, such as lists of internet addresses, or as less regular data such as

usernames. In the case that all the keys of a table are constant values, the table could be more efficiently implemented as a C struct. If a table is filled using only known strings, but can be accessed using strings read from the user, the table could profitably be implemented using perfect minimal hashing.

7.1.2 Foundations

Although higher level analyses for scripting languages, such as the table-based optimization in the last section, have not yet been developed, this dissertation provides the foundations on which they may be built. Knowing the types of each name in the program is significantly more difficult than previously anticipated, and pre-existing techniques—particularly those for more static languages—are severely lacking. Chapter 6 describes the significant work which provides the foundations for advanced analysis of PHP.

Even before creating an analysis framework, a compiler for PHP needed to be built. People have been writing ahead-of-time compilers for over 50 years, but the right set of techniques had not been developed to handle modern scripting languages. Chapter 5 describes the set of techniques used to build an ahead-of-time compiler for PHP. However, this led to significant difficulties along the way. Simple issues such as correctly managing memory and emulating ad-hoc calling conventions are difficult. Section 7.3 describes a better solution for scripting language implementation and analysis.

7.1.3 Future Work

Now that the foundations are laid, it is natural that others will build upon them. Promising ideas, such as table-based optimization, can now be built. A number of other optimizations have also appeared to be both necessary and straightforward to implement in the current optimization framework.

The number of dynamic types per PHP variable is low, but it could be significantly lower. Variables are often initialized implicitly in loops. By creating explicit initializations before loop entry, it should be possible to remove the *null* type from loop variables. This could possibly be performed automatically by peeling the first iteration off each loop. This is similar to *extended message splitting* [Chambers and Ungar, 1990], described in Section 2.2.1.

Numeric type results are often more conservative than may be required. Arithmetic operations between integers may result in floating point values, except in certain cases. These results can be constrained by using value-range propagation [Patterson, 1995]. This fits nicely into the symbolic execution framework described in Section 6.5.

I became enamoured with string-range propagation [Wassermann and Su, 2007], but never had time to implement it. Although it was designed for bug finding, I believe it can be used to optimize effectively. In particular, modelling the string values of a program can allow string operations to be simplified significantly. An interesting idea is to statically analyse regular expressions in order to optimize or compile them ahead-of-time.

Though the analyses in Section 6.5 are powerful, it is possible to make them more powerful still with small changes. One particularly useful change would be to model a *certainty* for field edges, in the same way as it is modelled for reference edges. This will allow us to know if a variable or field is already initialized at a certain point in the program, which can be used to optimize the generated code.

7.1.4 Experience of Working with PHP

Naturally, all of this assumes that more compiler researchers will become interested in PHP. In theory, I should support this. After spending so much time building the foundations, I should encourage other researchers to use my work as much as possible.

However, I have serious misgivings in encouraging others to work on, with or near PHP. In fact, I would encourage others to avoid it. I spent a significant amount of time working with the PHP language, the existing implementation, and the PHP community, and I could not recommend others to do the same.

The PHP language is inelegant and difficult to work with. This can clearly be seen from the behaviour of PHP described in Section 6.2. There are many edge cases which combine in awkward ways, which made implementing a program analysis a harrowing experience.

PHP was not designed; rather it grew upwards from a simple framework written nearly 15 years ago. It might be expected that this is a good thing, with many years to remove failed ideas, iterate on the design, and hone the language into an elegant tool for writing web applications. However, there is little evidence that this occurred, that

failed ideas were removed, or that the design was iterated. PHP's language features are those for which code was implemented and available, as opposed to those that were designed. The net result is a mishmash of features which do not work well together.

A good example is the PHP reference semantics. References were an important feature before objects existed, as the only other way to pass arrays was by copy. However, upon the addition of PHP objects, it became possible to pass pointers to objects, making references significantly less important. It would have been an elegant contribution to PHP to turn arrays into objects and remove references from the language. In particular, this would have allowed existing alias analyses to work on PHP with only minor tweaks. Instead, arrays continue to be treated differently to objects, and references were retained in the language, make analysis significantly more difficult. It took a significant amount of study, experimentation and effort to deal with PHP's references in a program analysis.

Despite this effort, the actual semantics of PHP are still undefined. The existing PHP implementation is the only thing to define them. During the study of the behaviour of PHP, in particular for the program analysis in Chapter 6, many of the semantics of PHP could only be derived from reading the C source code of the PHP system. For example, the types which could result from arithmetic statements, the behaviour of method invocations, and the behaviour of references in the presence of arrays, all needed significantly more detail that that provided by the manual. Unfortunately, my experience of reading the source of the PHP implementation tells me that the PHP implementation is low-quality. It comprises hundreds of thousands of lines of poorly written and documented code. It appears as if it was built by adding hacks to older hacks (it appears that PHP is currently being developed this way as well). As the code is the definition of the PHP language, each hack makes the definition more and more opaque. As such, working with the language is difficult and error-prone.

A large part of the difficulty of working with PHP stems from the PHP internals development community (that is, the community which develops the PHP implementation). When working on PHP, I chose to partake in this community, in order to understand how PHP works, how it is developed, and importantly the future direction of the PHP language. Ultimately, trying to work within the community is a harrowing experience. There is no clear leader in the community, senior members of the community do not seem to agree on how PHP should be developed, and there is a high level of attrition. The language features which a compiler researcher must understand are often created with little design and less discussion. The intended semantics are rarely documented in a thorough fashion, and—as with existing features—only

the source code describes the feature in sufficient detail to understand, analyse or implement. Peer review of new code and features is rarely offered, and more rarely is the peer review used.

The PHP user community is not significantly better. It is my experience that the PHP community does not care about its tools. Rather, it seems that many professional PHP developers are not interested in expanding beyond the niche of web development, resulting in little tool support. In other scripting language communities—Lua, Perl, Python, Ruby—tools are typically built by the user communities. Interested hackers, who love the elegance and beauty of their language of choice, build powerful tools such as LuaJit, Rubinius, Jython [Jython] and JRuby [JRuby]. This rarely occurs in PHP. A majority of interesting and useful tools for PHP have been instead been built by commercial companies or academics.

Finally, it seems the PHP internals community seems not to care about academic research. There have been few inroads of research into the PHP implementation. For example, the PHP interpreter is based on function calls, rather than switch statements, gotos, or more advanced techniques such as token threading. The PHP garbage collector is based on reference counting, and only recently has a simple mark-sweep collector been built (although reference counting still remains). By contrast, the PHP web site is well maintained and easy to navigate, the manual is filled with examples and notes from other users. The PHP community values good web design, and the ease with which PHP can be learned by new developers, but unfortunately this comes at a cost in the language design, implementation, and tool support.

Overall, my experience of working with PHP has not been positive. It is clear that there is a significant amount of research that remains to be done on the PHP language, and scripting languages in general. However, I could not encourage researchers to engage in this research on the PHP language or implementation.

7.2 Perhaps We're Going the Wrong Way

Ultimately, the work in this dissertation is aimed at solving real, practical problems in the compilation of PHP. All of these problems occur because the PHP language is not designed for ahead-of-time compilation. In fact, its not designed for any programming model. Rather, it has grown up around interpreters, and as a result its features take advantage of the interpreted environment.

In fact, it could reasonably be said that most scripting language research to date has been aimed at solving practical problems—practical problems which would not have existed if the language had been designed in a slightly different manner:

- Furr et al. [2009] partially solve the challenging problem that Ruby programs frequently call `eval` in their libraries, which impedes analysis. As `eval` statements are simple and available, they are frequently used for meta-programming. A more structured approach to meta-programming, such as Lisp's macros, would allow better expressiveness in meta-programming, and would no longer impede program analysis.

- Wassermann and Su [2007] find that special techniques are required to even find the list of source files in a program. There are very few programming languages in which simply finding the program text is such a complicated problem. A programming model which supported a different style of meta-programming would remove the use of run-time code generation, reducing the need for analysis.

- Gal et al. [2009] created new and powerful techniques for type feedback, to build JIT compiler for Javascript. At the same time, Jensen et al. [2009] show that Javascript variables almost never have more than one concrete type. If Javascript programs were typically available ahead-of-time, as is required for Jensen et al.'s technique, type-feedback would not be necessary for fast execution, making Gal et al.'s [2009] research unnecessary.

- Chapter 5 solves a problem by using C APIs. However, the real problem is the presence of the C APIs. They are merely a legacy of early extensions, which should have been quickly replaced by powerful foreign function interfaces, but never were. By using a declarative foreign-function-interface, discussed in Section 7.3, many of the techniques in Chapter 5 would be unnecessary.

- Chapter 5 also deals with the problem of languages which are only specified by their implementation. This problem would be avoided if languages were designed so that they could specified either in prose, or ideally with a formal definition.

- In Chapter 6 we developed a powerful alias analysis. One of the major features of the alias analysis is that it handles references correctly, which had been lacking in previous work, and was unnecessary for other languages. However, PHP references are a legacy feature; currently the only feature references provide

that cannot be provided in another way is to pass arrays by reference. If PHP had had object-oriented features from the start, then tables might have been objects (as they are in Python and Ruby), and PHP references would not be necessary. In turn, the complex alias analysis developed in Chapter 6 would not be necessary, and a much simpler analysis might have been devised.

- Our analysis from Chapter 6 also models that symbol-tables can be accessed as arrays. I do not know of any use of this feature that could be implemented in a cleaner manner.

These problems do not solely occur in scripting languages. Dynamic class-loading makes efficiently compiling a Java program ahead-of-time a difficult problem (and one which still requires an interpreter at run-time). As a result, researchers have spent 15 years trying to make fast JIT compilers so that Java can be nearly as fast as C++.

Most of the solutions to these problems are intractable for static analysis. However, none of these problems are due to important language features. Rather, these features are due to how these languages evolved, and how the provided features were used in practice.

Now that the features are present, they are impossible to remove. Using `eval` statements in libraries in Ruby has enabled a very powerful meta-programming technique called *monkey-patching*. Dynamic class-loading is an essential mechanism for Java plugins, such as those used in the Eclipse IDE. PHP's `eval` statement allows templating systems such as Smarty to create templates in PHP, which are then executed dynamically by the interpreter. Python and Perl programs often use the interpreter to execute Unix *.rc* files. Javascript's `eval` statements are widely used in web applications, to execute code fetched from an internet server, using a technique called *Asynchronous Javascript And XML* (AJAX).

However, each of these features could have been implemented in a way that did not damage static analysis. If monkey-patching had never existed, it is unlikely that there would be a call for it. C programs can use plugins without significantly ruining opportunities for optimization. Some of the techniques listed above simply leverage the interpreter to execute an almost completely different program, which could easily be sandboxed in such a way as to enable static analysis.

All of these techniques are a means of encoding program data as program code. This is very popular in Lisp, but Lisp at least has a structured technique to perform this.

Here, the strings that are `eval`'ed are relatively arbitrary. In order to facilitate advanced program analysis of scripting languages, it is necessary to remove these techniques, replacing them with structured and analysable facilities.

7.2.1 Are JIT Compilers the Future?

After over 20 years of scripting languages, in which they have risen to be very important and popular languages, there has not been a mainstream ahead-of-time compiler for any scripting language. However, there are a number of JIT compilers which were recently built or developed: Parrot, Unladen-swallow, Rubinius, LuaJit, TraceMonkey, V8 and Tamarin. It seems that the scripting language community is moving in this direction.

However, there are many features of ahead-of-time compilers which are not available in JIT compilers. Currently, the major direction of JIT research for dynamic languages are to remove the cost of dynamic typing. Means of providing expensive optimizations such as partial redundancy elimination or loop-invariant code motion during JIT compilation have yet to be explored. Advanced techniques such as those described in Section 7.1.1 are also expensive, and less likely to be made available to JIT compilers.

I feel this is regrettable. In Section 7.1.1 I lamented the problems of optimizing low-level languages. However, it is clear that high-level scripting languages in their current form do not help solve this problem. If JIT compilers are to become the sole means of optimizing dynamic scripting languages, then producing a fast program will remain a great deal of work for the programmer.

7.3 A Better Place

Clearly the designs of scripting languages create problems for program analysis. In this section, I outline features that future scripting languages should consider, in order to enable both the expressiveness and freedom currently provided by scripting languages, as well as the benefits of program analysis.[1]

- In my opinion, the major feature of a scripting language is that it is dynamically typed and duck-typed, with tables being the major unit of data structuring,

[1] I note that many of these features are available in the Factor programming language.

including objects. In existing scripting languages, these are powerful and expressive features. In my experience, when users of scripting languages discuss the features which are important to them, and why they enjoy developing in scripting languages, the features they cite exist because of these features. All other features, especially `eval` statements and C APIs, appear to be incidental to the programming experience and should be omitted, as explained below.

- The language should have a prose specification, and avoid having a canonical implementation. If possible, there should be multiple implementations, include a reference implementation whose design is as simple as possible, and only intended for verifying a program's semantics. Languages should also avoid features which restrict how the language may be implemented. A C API is an extreme example of such a restriction. Another example is a Python feature which allows the call-stack to be introspected directly. These features restrict reimplementations of the language, and should either be omitted from the language, or their exact semantics should not be proscribed.

- The foreign-function interface should not be based on any implementation of the language. Rather, it should be declarative, and the mechanism of the interface should be automatically generated from the declarations. A simple example is to declare C functions, which can be automatically made available as functions in the scripting language.

  Even languages designed for embedding, such as Lua [Ierusalimschy et al., 2007], can expose their implementations. To call a Lua function from C, parameters are pushed onto a program stack. If this became the bottleneck in Lua programs, it would be impossible to remove it without breaking compatibility with old programs. By contrast, a declarative interface would require at most a recompile of the C program.

- The `include` statement should be separated according to whether it is static or dynamic. Static includes are program source, and knowing the program source at compile-time makes compilation and analysis significantly easier. In general, it would be useful to split dynamic features into separate compile-time and run-time features.

- The `eval` statement is unnecessary. Many features that it supports through meta-programming can be done equally well or better at compile-time. It it likely to be safer to execute `.rc` files in a sandboxed environment, rather than directly. Plugins are likely to require more research—they could easily break type inference—but a variant of SafeTSA [Amme et al., 2001] could be used.

- There seems to be no good reason for variable-variables.
- Every value should be an object. A separation between primitives, arrays and objects leads to a need for PHP-style references, as does a default of copying an array by value. Operators should be methods. If a language allows implicit conversions, such as `NULL + 5` or any indexing or operators of undefined values, should be implemented as operators of the NULL type. This will help avoid the sort of implicit semantics that authors of reimplementations and program analyses struggle to replicate.
- The standard libraries should be written in the language itself. As low-level libraries such as string manipulation and hashtable access will then be executed by the language implementation itself, this provides a strong impetus to make the implementation fast.
- In my experience, having a single person in charge of the language—a so called *Benevolent Dictator For Life*[2]—seems to lead to elegant, expressive languages.

7.4 Contribution

The dissertation makes several contributions to the study of compilers for scripting languages. The two primary contributions are a description of how to create an ahead-of-time compiler for scripting languages, discussed in Section 7.4.1, and how to statically analyse PHP, discussed in Section 7.4.2. A secondary contribution is an extensive discussion of PHP and scripting language semantics, discussed in Section 7.4.3. Finally, an artefact of the dissertation is the `phc` compiler, discussed in Section 7.4.4. Overall, these contributions combine to demonstrate my thesis: that '*compiling scripting languages using an ahead-of-time compiler is possible and valuable*'. In this section, I discuss the effectiveness of the contributions, identifying particular advantages and weaknesses.

7.4.1 Scripting Language Compilation

Chapter 5 explains how to effectively compile scripting languages ahead-of-time. A major contribution of this chapter was identifying the problems which make

[2] *Benevolent Dictator For Life* is a name given to Guido van Rossum, the head of the Python project.

compiling scripting languages difficult. There are three major challenges, C APIs, run-time code generation and undefined language semantics. The solution is to integrate the canonical implementation into both the compiler and the generated code.

The technique is most effective for dealing with limited run-time code generation. Important uses are when a program uses an `eval` to read a configuration file, and to execute plugins or localisation files. In my experience, these are the most prevalent forms of run-time code generation in PHP. However, if the programmer uses dynamic features to create large portions of their application, then the benefits of compilation are not available, as the application is effectively being interpreted. In some circumstances, this may constrain the design of the application. In these rare cases, the application could simply be interpreted.

We show how to compile a scripting language to support run-time code generation, generating code using the C API. The downside is that this constrains the compiler. Ideally, a compiler would be able to produce code which runs a program many times faster than an interpreter. Unfortunately, our speedup is closer to 1.5x on average. However, the compiled model allows the use of static analysis, and the optimizer discussed in Section 6 should allow great speed improvements in the future.

We handle the undefined language features by integrating the PHP system into `phc`. This protects us from a large number of changes in the PHP language. However, it is impossible to know how the language may change in the future, and this may require changes to the compiler in the future. In addition, we are not protected from changes in the PHP syntax. However, in general, this technique is very effective, especially in the optimizer, and the integration is very simple and straightforward to implement.

7.4.2 Static Analysis

Chapter 6 describes an algorithm to provide effective static analysis for PHP. It combines alias analysis, type-inference, and constant propagation, and effectively handles a significant portion of PHP's language features. It is able to prove that the PHP programs we have analysed have very few types per variable—typically only one—and are almost never aliased.

The alias analysis is a particular contribution. PHP has very unusual semantics, and the run-time references are not a feature of any language I have seen statically

analysed.[3] The alias analysis is therefore specifically crafted to support both may- and must-aliases between run-time values. It supports arrays, tables, objects and variables with the same precision, falling back gracefully to a less precise solution when less information is present. The analysis is field-sensitive, object-sensitive, flow-sensitive and context-sensitive, though the last two are configurable.

The downside of the context- and flow-sensitivity is that the analysis is not fast. It was not designed for speed or scalability. While nothing in its design should impede scalability, we do not have an evidence of how it scales to large programs. Certainly, it seems that it should be possible it combine the analysis with existing techniques for scalable analysis [Pearce et al., 2007, Whaley and Lam, 2004].

A concern is that the analysis does not combine naturally with run-time code generation. Chapter 5 describes how `phc` handles run-time code generation, and it is unfortunate that we can support it for compilation, but not analysis. I believe that future work may allow the analysis to work up until the point in the program where run-time code generation is used. Alternatively, there may be a means of using annotation to allow the analysis to co-exist with run-time code generation. I hope to see future work in this area.

SSA is a powerful intermediate representation on which to build program analysis. Section 6.5.5 describes how to build SSA form for a language such as PHP. One concern is that SSA cannot be used while performing the alias analysis. This is an open problem in general. However, I have taken the first step to making this work, using the algorithm in Section 4.6.4. This allows SSA form to be built on-demand using a symbolic execution framework—the same framework on which our alias analysis is built. Hopefully, this will lead to a solution for the phase ordering problem between SSA form and alias analysis.

7.4.3 Scripting Language Behaviour

A major contribution is identifying the behaviour of PHP, especially in relation to compilation and program analysis. Chapters 5 and 2 identify common patterns and components of the design of important scripting languages. Section 6.2 goes into great detail on the behaviour of PHP. In addition, Section 6.3 identifies how previous analyses for PHP did not correctly identify the behaviour of PHP features, and how

[3] Perl also supports run-time references, but there appears to be no literature discussing alias analysis for Perl.

that affected their analysis.

7.4.4 The phc Compiler

A major artefact of this research is the phc compiler, an open source compiler for PHP. Chapter 4 describes important aspects of its design. Its compilation technique is described in Chapter 5, and its optimizer is described in Chapter 6. As phc is open source, and available online, it may be used for future research into PHP, in particular for static analysis. In fact, it has been used for some research already [Wassermann et al., 2008].

7.5 Conclusion

Throughout this dissertation, it has been shown how ahead-of-time compilers for scripting languages are possible to construct and valuable to have. That compilation is is possible is demonstrated in Chapter 5. That program analysis for dynamic scripting languages is valuable is shown in Chapter 6. This chapter has summarised the contributions of this dissertation. Overall, I believe that my thesis has been shown: that *compiling scripting languages using an ahead-of-time compiler is possible and valuable*.

Intermediate Representation Definitions

Each of phc's *Intermediate Representations* (IRs) is defined using an abstract grammar. The abstract grammars are used to create C++ classes to represent the IRs and to provide APIs for visitation and transformation. `maketea` [de Vries and Gilbert, 2007, 2008] was developed to automatically create these classes from grammar specifications.

This appendix shows three intermediate representations: the *Abstract Syntax Tree*, the *High-level Intermediate Representation* and the *Medium-level Intermediate Representation*. They are described in more detail in Section 4.2.

A.1 Abstract Syntax Tree

```
{-
   Top-level structure
-}

PHP_script ::= Statement* ;

Statement ::=
     Class_def | Interface_def | Method
    | Return | Static_declaration | Global
    | Try | Throw | Eval_expr
    | Break | Continue
    | If | Foreach
    | While | Do | For | Switch
    | Declare
    ;

{-
    Class and method definitions
-}

Class_def ::=
   Class_mod CLASS_NAME extends:CLASS_NAME? implements:INTERFACE_NAME* Member* ;
Class_mod ::= "abstract"? "final"? ;

Interface_def ::= INTERFACE_NAME extends:INTERFACE_NAME* Member* ;

Member ::= Method | Attribute ;
```

```
Method ::= Signature Statement*? ;
Signature ::= Method_mod return_by_ref:"&"? METHOD_NAME Formal_parameter* ;
Method_mod ::= "public"? "protected"? "private"? "static"? "abstract"? "final"? ;
Formal_parameter ::= Type pass_by_ref:"&"? var:Name_with_default ;
Type ::= CLASS_NAME? ;

Attribute ::= Attr_mod vars:Name_with_default* ;
Attr_mod ::= "public"? "protected"? "private"? "static"? "const"?  ;

Name_with_default ::= VARIABLE_NAME Expr? ;


{-
   Statements
-}

Return ::= Expr? ;

Static_declaration ::= vars:Name_with_default* ;
Global ::= Variable_name* ;

Try ::= Statement* catches:Catch* ;
Catch ::= CLASS_NAME VARIABLE_NAME Statement* ;
Throw ::= Expr ;

Eval_expr ::= Expr ;

{-
 - Pre-MIR statements
 -}
If ::= Expr iftrue:Statement* iffalse:Statement* ;
Foreach ::= Expr key:Variable? is_ref:"&"? val:Variable Statement* ;

Break ::= Expr? ;
Continue ::= Expr? ;


{-
 - AST-only statements
 -}

Declare ::= Directive+ Statement* ;
Directive ::= DIRECTIVE_NAME Expr ;

While ::= Expr Statement* ;
Do ::= Statement* Expr ;
For ::= init:Expr? cond:Expr? incr:Expr? Statement* ;
Switch ::= Expr Switch_case* ;
Switch_case ::= Expr? Statement* ;
```

```
{-
   Expressions
-}

Expr ::=
      Literal
    | Cast | Unary_op | Bin_op
    | Constant | Instanceof
    | Method_invocation | New
    | Variable
    | Assignment | Op_assignment | List_assignment
    | Pre_op | Post_op
    | Array | Conditional_expr | Ignore_errors
    ;

Literal ::= INT | REAL | STRING | BOOL | NIL ;

Cast     ::= CAST Expr ;
Unary_op ::= OP Expr ;
Bin_op   ::= left:Expr OP right:Expr ;

Constant ::= CLASS_NAME? CONSTANT_NAME ;
Instanceof ::= Expr Class_name ;

Method_invocation ::= Target? Method_name Actual_parameter* ;
New ::= Class_name Actual_parameter* ;

Actual_parameter ::= pass_by_ref:"&"? Expr ;
Target ::= Expr | CLASS_NAME ;


{-
 - AST-only expressions
 -}

Variable ::= Target? Variable_name array_indices:Expr?* ;

Assignment ::= Variable is_ref:"&"? Expr ;
Op_assignment ::= Variable OP Expr ;
List_assignment ::= List_element?* Expr ;
List_element ::= Variable | Nested_list_elements ;
Nested_list_elements ::= List_element?* ;
Array ::= Array_elem* ;
Array_elem ::= key:Expr? is_ref:"&"? val:Expr ;

Pre_op ::= OP Variable ;
Post_op ::= Variable OP ;

Conditional_expr ::= cond:Expr iftrue:Expr iffalse:Expr ;
Ignore_errors ::= Expr ;
```

```
-- Identifiers
Method_name ::= METHOD_NAME | Reflection ;
Class_name ::= CLASS_NAME | Reflection ;
Variable_name ::= VARIABLE_NAME | Reflection ;

Reflection ::= Expr ;
```

Listing A.1: *Language definition of the* 'Abstract Syntax Tree'*, in* `maketea` *format. The definition omits several declarations used for engineering purposes, that do not have a bearing on the language itself.*

A.2 High-level Intermediate Representation

```
{-
   Top-level structure
-}

PHP_script ::= Statement* ;

Statement ::=
     Class_def | Interface_def | Method
    | Return | Static_declaration | Global
    | Try | Throw | Eval_expr
    | Break | Continue
    | If | Foreach
    | Loop
    | Assign_var | Assign_var_var | Assign_array | Assign_next | Assign_field
    | Pre_op
    ;

{-
    Class and method definitions
-}

Class_def ::=
   Class_mod CLASS_NAME extends:CLASS_NAME? implements:INTERFACE_NAME* Member* ;
Class_mod ::= "abstract"? "final"? ;

Interface_def ::= INTERFACE_NAME extends:INTERFACE_NAME* Member* ;

Member ::= Method | Attribute ;

Method ::= Signature Statement*? ;
Signature ::= Method_mod return_by_ref:"&"? METHOD_NAME Formal_parameter* ;
Method_mod ::= "public"? "protected"? "private"? "static"? "abstract"? "final"? ;
Formal_parameter ::= Type pass_by_ref:"&"? var:Name_with_default ;
Type ::= CLASS_NAME? ;

Attribute ::= Attr_mod var:Name_with_default ;
Attr_mod ::= "public"? "protected"? "private"? "static"? "const"?  ;

Name_with_default ::= VARIABLE_NAME default_value:Static_value? ;

{-
   Statements
-}

Return ::= Rvalue ;

Static_declaration ::= var:Name_with_default ;
Global ::= Variable_name ;

Try ::= Statement* catches:Catch* ;
Catch ::= CLASS_NAME VARIABLE_NAME Statement* ;
```

```
Throw ::= VARIABLE_NAME ;

Eval_expr ::= Expr ;

{-
 - Pre-MIR statements
 -}
If ::= VARIABLE_NAME iftrue:Statement* iffalse:Statement* ;
Foreach ::= arr:VARIABLE_NAME key:VARIABLE_NAME? is_ref:"&"? val:VARIABLE_NAME
    Statement* ;

Break ::= Expr? ;
Continue ::= Expr? ;


{-
 - Post-AST statements
 -}

Assign_var     ::=          lhs:VARIABLE_NAME
    is_ref:"&"? rhs:Expr ;
Assign_field    ::= Target  Field_name
    is_ref:"&"? rhs:Rvalue ;
Assign_array   ::=          lhs:VARIABLE_NAME index:Rvalue          is_ref:"&"?
    rhs:Rvalue ;
Assign_var_var ::=          lhs:VARIABLE_NAME
    is_ref:"&"? rhs:Rvalue ;
Assign_next     ::=         lhs:VARIABLE_NAME
    is_ref:"&"? rhs:Rvalue ;
Pre_op ::= OP VARIABLE_NAME ;

{-
 - HIR-only statements
 -}

Loop ::= Statement* ;


{-
   Expressions
-}

Expr ::=
      Literal
    | Cast | Unary_op | Bin_op
    | Constant | Instanceof
    | Method_invocation | New
    | Variable_name | Array_access | Field_access | Array_next
    ;

Literal ::= INT | REAL | STRING | BOOL | NIL ;

Cast     ::= CAST VARIABLE_NAME ;
```

```
Unary_op ::= OP VARIABLE_NAME ;
Bin_op   ::= left:Rvalue OP right:Rvalue ;

Constant ::= CLASS_NAME? CONSTANT_NAME ;
Instanceof ::= VARIABLE_NAME Class_name ;

Method_invocation ::= Target? Method_name Actual_parameter* ;
New ::= Class_name Actual_parameter* ;

Actual_parameter ::= pass_by_ref:"&"? Rvalue ;
Target ::= VARIABLE_NAME | CLASS_NAME ;

{-
 - Post-AST expressions
 -}

Rvalue ::= Literal | VARIABLE_NAME ;

Field_access ::= Target Field_name ;
Array_access ::= VARIABLE_NAME index:Rvalue ;
Array_next   ::= VARIABLE_NAME ;

Static_value ::= Literal | Static_array | Constant ;
Static_array      ::= Static_array_elem* ;
Static_array_elem ::= key:Static_array_key? is_ref:"&"? val:Static_value ;
Static_array_key  ::= Literal | Constant ;


{-
 - Idenfifiers
 -}
Method_name ::= METHOD_NAME | Variable_method ;
Variable_name ::= VARIABLE_NAME | Variable_variable ;
Class_name ::= CLASS_NAME | Variable_class ;
Field_name ::= FIELD_NAME | Variable_field ;

Variable_method ::= VARIABLE_NAME ;
Variable_variable ::= VARIABLE_NAME ;
Variable_class ::= VARIABLE_NAME ;
Variable_field ::= VARIABLE_NAME ;
```

Listing A.2: *Language definition of the* 'High-level Intermediate Representation'*, in* `maketea` *format. The definition omits several declarations used for engineering purposes, that do not have a bearing on the language itself.*

A.3 Medium-level Intermediate Representation

```
{-
   Top-level structure
-}

PHP_script ::= Statement* ;

Statement ::=
     Class_def | Interface_def | Method
    | Return | Static_declaration | Global
    | Try | Throw | Eval_expr
    | Assign_var | Assign_var_var | Assign_array | Assign_next | Assign_field
    | Label | Goto | Branch
    | Foreach_next | Foreach_reset | Foreach_end
    | Class_alias | Interface_alias | Method_alias
    | Pre_op
    | Unset
    ;

{-
    Class and method definitions
-}

Class_def ::=
   Class_mod CLASS_NAME extends:CLASS_NAME? implements:INTERFACE_NAME* Member* ;
Class_mod ::= "abstract"? "final"? ;

Interface_def ::= INTERFACE_NAME extends:INTERFACE_NAME* Member* ;

Member ::= Method | Attribute ;

Method ::= Signature Statement*? ;
Signature ::= Method_mod return_by_ref:"&"? METHOD_NAME Formal_parameter* ;
Method_mod ::= "public"? "protected"? "private"? "static"? "abstract"? "final"? ;
Formal_parameter ::= Type pass_by_ref:"&"? var:Name_with_default ;
Type ::= CLASS_NAME? ;

Attribute ::= Attr_mod var:Name_with_default ;
Attr_mod ::= "public"? "protected"? "private"? "static"? "const"? ;

Name_with_default ::= VARIABLE_NAME default_value:Static_value? ;

{-
 - MIR-only dynamic definitions
 -}
Class_alias ::= alias:CLASS_NAME CLASS_NAME ;
Interface_alias ::= alias:INTERFACE_NAME INTERFACE_NAME ;
Method_alias ::= alias:METHOD_NAME METHOD_NAME ;

{-
   Statements
-}
```

```
Return ::= Rvalue ;

Static_declaration ::= var:Name_with_default ;
Global ::= Variable_name ;

Try ::= Statement* catches:Catch* ;
Catch ::= CLASS_NAME VARIABLE_NAME Statement* ;
Throw ::= VARIABLE_NAME ;

Eval_expr ::= Expr ;

{-
 - Post-AST statements
 -}

Assign_var     ::=          lhs:VARIABLE_NAME
    is_ref:"&"? rhs:Expr ;
Assign_field    ::= Target  Field_name
    is_ref:"&"? rhs:Rvalue ;
Assign_array   ::=          lhs:VARIABLE_NAME index:Rvalue          is_ref:"&"?
    rhs:Rvalue ;
Assign_var_var ::=          lhs:VARIABLE_NAME
    is_ref:"&"? rhs:Rvalue ;
Assign_next     ::=         lhs:VARIABLE_NAME
    is_ref:"&"? rhs:Rvalue ;
Pre_op ::= OP VARIABLE_NAME ;

{-
 - MIR-only statements
 -}

Unset ::= Target? Variable_name array_indices:Rvalue* ;

{-
   Expressions
-}

Expr ::=
      Literal
    | Cast | Unary_op | Bin_op
    | Constant | Instanceof
    | Method_invocation | New
    | Variable_name | Array_access | Field_access | Array_next
    | Isset
    | Foreach_has_key | Foreach_get_key | Foreach_get_val
    | Param_is_ref
    ;

Literal ::= INT | REAL | STRING | BOOL | NIL ;

Cast     ::= CAST VARIABLE_NAME ;
Unary_op ::= OP VARIABLE_NAME ;
```

```
Bin_op   ::= left:Rvalue OP right:Rvalue ;

Constant ::= CLASS_NAME? CONSTANT_NAME ;
Instanceof ::= VARIABLE_NAME Class_name ;

Method_invocation ::= Target? Method_name Actual_parameter* ;
New ::= Class_name Actual_parameter* ;

Actual_parameter ::= pass_by_ref:"&"? Rvalue ;
Target ::= VARIABLE_NAME | CLASS_NAME ;

{-
 - Post-AST expressions
 -}
Rvalue ::= Literal | VARIABLE_NAME ;

Field_access ::= Target Field_name ;
Array_access ::= VARIABLE_NAME index:Rvalue ;
Array_next   ::= VARIABLE_NAME ;{haskell}


-- Post-AST arrays
Static_value ::= Literal | Static_array | Constant ;
Static_array      ::= Static_array_elem* ;
Static_array_elem ::= key:Static_array_key? is_ref:"&"? val:Static_value ;
Static_array_key  ::= Literal | Constant ;


{-
 - MIR-only expressions
 -}

Isset ::= Target? Variable_name array_indices:Rvalue* ;

Foreach_reset ::= array:VARIABLE_NAME iter:HT_ITERATOR ;
Foreach_next ::= array:VARIABLE_NAME iter:HT_ITERATOR ;
Foreach_end ::= array:VARIABLE_NAME iter:HT_ITERATOR ;
Foreach_has_key ::= array:VARIABLE_NAME iter:HT_ITERATOR ;
Foreach_get_key ::= array:VARIABLE_NAME iter:HT_ITERATOR ;
Foreach_get_val ::= array:VARIABLE_NAME iter:HT_ITERATOR ;

Branch ::= VARIABLE_NAME iftrue:LABEL_NAME iffalse:LABEL_NAME;
Goto ::= LABEL_NAME ;
Label ::= LABEL_NAME ;

Param_is_ref ::= Target? Method_name PARAM_INDEX<int> ;


{-
 - Idenfifiers
 -}
Method_name ::= METHOD_NAME | Variable_method ;
Variable_name ::= VARIABLE_NAME | Variable_variable ;
```

```
Class_name ::= CLASS_NAME | Variable_class ;
Field_name ::= FIELD_NAME | Variable_field ;

Variable_method ::= VARIABLE_NAME ;
Variable_variable ::= VARIABLE_NAME ;
Variable_class ::= VARIABLE_NAME ;
Variable_field ::= VARIABLE_NAME ;
```

Listing A.3: *Language definition of the* 'Medium-level Intermediate Representation'*, in* `maketea` *format. The definition omits several declarations used for engineering purposes, that do not have a bearing on the language itself.*

Whole Program Optimization Interface

The phc whole program analysis framework is structured as a driver analysis with a number of client sub-analyses. In order to pass optimization information to client sub-analyses, the following API is used:

- `create_reference`: create references between two names,
- `assign_value`: set the value of a name to a storage node (see Section 6.5.2),
- `set_storage`: set the type of a storage node,
- `set_scalar`: set the value of a storage node used for scalars,
- `kill_value`: remove existing value for a name
- `record_use`: record that a variable or name is used (for *use-def analysis*)

In addition, the optimization framework needs to query the client analyses in order to:

- get a list of possible or definite aliases for a variable, field or array index,
- get the type of an object,
- get the value of an array or string offset, or of a variable-variable, -field, -method or -class,
- determine if the client analysis' results have changed, requiring further analysis.

Optimization Transformations

The phc compiler performs a small number of optimizations based on the results of our static analysis:

- branches with a known direction are replaced with direct edges to the known target,
- literals are propagated into any place in which they are used in a read-only capacity, such as in an array index,
- calls to pure methods are removed, if their result is a literal,
- expressions with known literal results are replaced with that literal,
- simple functions can be inlined at monomorphic call-sites.

A small number of optimizations are not performed, but would be straightforward:

- unnecessary casts can be removed
- unset statements (see Section 4.2.4) can be removed, if the variable is not set
- variable-variables, -fields, -methods and -classes (see Section 2.1.2) can be used directly.

In addition, it is straightforward to use the analysis results to generate better code, though this is not yet implemented:

- generally, each read and write of a variables must check if the variable is initialized, but we know this statically in many cases,
- type information can be used to remove type checks,
- variables with a single static type can be accessed through the C data type, potentially using a single register instead of a boxed value (this is called *scalar replacement of aggregates* [Muchnick, 1997]).

Bibliography

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26. Springer-Verlag, 1995. (Cited on pages 36, 41 and 144.)

[2] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–107. ACM, 1995. URL http://doi.acm.org/10.1145/217838.217847. (Cited on pages 14, 15, 16 and 37.)

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley Publishing Company, USA, 2007. ISBN 0321547985, 9780321547989. (Cited on page 20.)

[4] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48. ACM, 2007. URL http://doi.acm.org/10.1145/1251535.1251543. (Cited on page 29.)

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM, 1988. URL http://doi.acm.org/10.1145/73560.73561. (Cited on page 30.)

[6] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. Safetsa: a type safe and referentially secure mobile-code representation based on static single assignment form. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 137–147. ACM, 2001. doi: http://doi.acm.org/10.1145/378795.378825. (Cited on page 155.)

[7] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks, 2002. (Cited on page 137.)

[8] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, 1994. (Cited on pages 22, 24, 25, 28, 29, 36, 37, 43 and 144.)

[9] J. Aycock. Aggressive type inference. In *The Eighth International Python Conference*, 2000. URL http://www.python.org/workshops/2000-01/proceedings/papers/aycock/aycock.html. (Cited on page 40.)

[10] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2): 97–113, 2003. ISSN 0360-0300. (Cited on pages 14 and 88.)

[11] John Aycock. Converting Python virtual machine code to C. In *Proceedings of the 7th International Python Conference*, 1998. (Cited on pages 76, 78 and 91.)

[12] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 110–124. Springer-Verlag, 2000. (Cited on page 31.)

[13] John Aycock, David Pereira, and Georges Jodoin. UCPy: Reverse engineering Python. In *PyCon DC2003*, March 2003. (Cited on pages 75 and 78.)

[14] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341. ACM, 1996. URL http://doi.acm.org/10.1145/236337.236371. (Cited on page 35.)

[15] Yosi Ben Asher and Nadav Rotem. The effect of unrolling and inlining for python bytecode optimizations. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–14. ACM, 2009. URL http://doi.acm.org/10.1145/1534530.1534550. (Cited on page 43.)

[16] Jan Benda, Tomas Matousek, and Ladislav Prosek. Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. In *.NET Technologies 2006*, May 2006. (Cited on pages 76, 77 and 78.)

[17] Daniel Berlin. Structure aliasing in GCC. In *Proceedings of the GCC Developers' Summit*, pages 25–36, 2005. (Cited on page 29.)

[18] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114. ACM, 2003. URL http://doi.acm.org/10.1145/781131.781144. (Cited on page 29.)

[19] Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1916–1923. ACM, 2009. doi: http://doi.acm.org/10.1145/1529282.1529709. (Cited on page 4.)

[20] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM, 1999. URL http://doi.acm.org/10.1145/320384.320387. (Cited on page 26.)

[21] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. *SIGPLAN Not.*, 34(10):35–46, 1999. URL http://doi.acm.org/10.1145/320385.320388. (Cited on page 26.)

[22] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, 1994. doi: http://doi.acm.org/10.1145/197320.197331. (Cited on page 31.)

[23] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, Jul 1998. (Cited on page 31.)

[24] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. URL http://doi.acm.org/10.1145/136035.136043. (Cited on page 29.)

[25] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250. Springer-Verlag, 1994. (Cited on pages 23, 25, 39 and 116.)

[26] Brett Cannon. Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University, 2005. (Cited on pages 41 and 144.)

[27] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004. (Cited on page 12.)

[28] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 146–160. ACM, 1989. URL http://doi.acm.org/10.1145/73141.74831. (Cited on pages 14 and 15.)

[29] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 49–70. ACM, 1989. URL http://doi.acm.org/10.1145/74877.74884. (Cited on page 15.)

[30] Craig Chambers and David Ungar. Interactive type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 150–164. ACM, 1990. URL http://doi.acm.org/10.1145/93542.93562. (Cited on pages 15 and 148.)

[31] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310. ACM, 1990. URL http://doi.acm.org/10.1145/93542.93585. (Cited on pages 21, 22, 29 and 32.)

[32] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1993. URL http://doi.acm.org/10.1145/158511.158639. (Cited on pages 23, 24 and 25.)

[33] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Incremental computation of static single assignment form. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 223–237. Springer-Verlag, 1996. (Cited on page 31.)

[34] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, 1999. ISSN 0362-1340. URL http://doi.acm.org/10.1145/320385.320386. (Cited on pages 26, 27, 28, 119 and 126.)

[35] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003. (Cited on pages 25 and 28.)

[36] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 273–286. ACM, 1997. URL http://doi.acm.org/10.1145/258915.258940. (Cited on page 30.)

[37] 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*, pages 253–267. Springer-Verlag, 1996. (Cited on pages 32, 63 and 132.)

[38] Rezaul Alam Chowdhury, Peter Djeu, Brendon Cahoon, James H. Burrill, and Kathryn S. McKinley. The limits of alias analysis for scalar optimizations. In *CC '04: Proceedings of the 13th International Conference on Compiler Construction*, pages 24–38. Springer, 2004. (Cited on page 33.)

[39] A.S. Christensen, A. Møller, and M.I. Schwartzbach. Precise analysis of string expressions. In *SAS'03: 10th International Static Analysis Symposium*, pages 1–18. Springer-Verlag, 2003. (Cited on page 43.)

[40] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, 1995. URL http://doi.acm.org/10.1145/201059.201061. (Cited on pages 38 and 39.)

[41] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 57–66. ACM, 1988. URL http://doi.acm.org/10.1145/53990.53996. (Cited on page 22.)

[42] K. D. Cooper and K. Kennedy. Fast interprocedual alias analysis. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of*

programming languages, pages 49–59. ACM, 1989. URL http://doi.acm.org/10.1145/75277.75282. (Cited on page 22.)

[43] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989. URL http://doi.acm.org/10.1145/75277.75280. (Cited on page 30.)

[44] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. *SIGPLAN Not.*, 28(6):36–45, 1993. URL http://doi.acm.org/10.1145/173262.155094. (Cited on pages 21 and 33.)

[45] 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. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925. URL http://doi.acm.org/10.1145/115372.115320. (Cited on pages 30, 31 and 32.)

[46] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982. URL http://doi.acm.org/10.1145/582153.582176. (Cited on page 37.)

[47] Dibyendu Das and U. Ramakrishna. A practical and fast iterative algorithm for ϕ-function computation using dj graphs. *ACM Trans. Program. Lang. Syst.*, 27 (3):426–440, 2005. doi: http://doi.acm.org/10.1145/1065887.1065890. (Cited on page 31.)

[48] Edsko de Vries and John Gilbert. Design and Implementation of a PHP Compiler Front-end. Dept. of Computer Science Technical Report TR-2007-47, Trinity College Dublin, September 2007. (Cited on pages 3, 47 and 161.)

[49] Edsko de Vries and John Gilbert. Processing ASTs in C++: maketea. 2008. URL http://phpcompiler.org/doc/maketea.pdf. (Cited on pages 3, 47 and 161.)

[50] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, 1995. (Cited on pages 35 and 137.)

[51] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 106–117. ACM, 1998. doi: http://doi.acm.org/10.1145/277650.277670. (Cited on pages 26, 119 and 132.)

[52] Mark Dufour. Shed Skin: An optimizing Python-to-C++ compiler. Master's thesis, Delft University of Technology, 2006. (Cited on pages 41, 76, 77, 78 and 144.)

[53] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256. ACM, 1994. doi: http://doi.acm.org/10.1145/178243.178264. (Cited on pages 21, 24, 25, 27, 31, 116, 118, 123 and 124.)

[54] Greg Ewing. Pyrex - a language for writing Python extension modules. URL `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`. (Cited on pages 77 and 78.)

[55] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 85–96. ACM, 1998. URL `http://doi.acm.org/10.1145/277650.277667`. (Cited on page 28.)

[56] Marc Feeley. Speculative inlining of predefined procedures in an R5RS Scheme to C compiler. pages 237–253. Springer-Verlag, 2007. doi: http://dx.doi.org/10.1007/978-3-540-85373-2_14. (Cited on pages 73, 96 and 98.)

[57] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174. Springer-Verlag, 2000. (Cited on page 34.)

[58] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1859–1866. ACM, 2009. URL `http://doi.acm.org/10.1145/1529282.1529700`. (Cited on pages 42 and 144.)

[59] Michael Furr, Jong hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2009. (Cited on pages 42, 102, 114, 136 and 152.)

[60] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153. ACM, 2006. (Cited on page 88.)

[61] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478. ACM, 2009. doi: http://doi.acm.org/10.1145/1542476.1542528. (Cited on pages 17, 88, 96, 141 and 152.)

[62] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 82–93. Springer-Verlag, 2000. (Cited on page 26.)

[63] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15. ACM, 1996. URL `http://doi.acm.org/10.1145/237721.237724`. (Cited on page 29.)

[64] Sara Golemon. *Extending and Embedding PHP*. Sams, 2006. ISBN 067232704X. (Cited on page 80.)

[65] Richard Guenther. GIMPLE alias improvements for GCC 4.5. In *Proceedings of the GCC Developers' Summit*, pages 47–56, June 2009. (Cited on page 34.)

[66] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-me: a static analysis for automatic individual object reclamation. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language*

design and implementation, pages 364–375. ACM Press, 2006. (Cited on page 28.)

[67] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 69–80. ACM, 2006. URL http://doi.acm.org/10.1145/1181775.1181785. (Cited on page 29.)

[68] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 290–299. ACM, 2007. (Cited on page 29.)

[69] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 226–238. ACM, 2009. URL http://doi.acm.org/10.1145/1480881.1480911. (Cited on page 29.)

[70] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 254–263. ACM, 2001. URL http://doi.acm.org/10.1145/378795.378855. (Cited on page 29.)

[71] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123. ACM, 2000. URL http://doi.acm.org/10.1145/347324.348916. (Cited on pages 21, 23 and 25.)

[72] Urs Hölzle and David Ungar. A third-generation Self implementation: reconciling responsiveness with performance. *SIGPLAN Not.*, 29(10):229–243, 1994. (Cited on page 15.)

[73] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991. (Cited on page 15.)

[74] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *SIGPLAN Not.*, 24(7):28–40, 1989. ISSN 0362-1340. URL http://doi.acm.org/10.1145/74818.74821. (Cited on page 22.)

[75] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004. doi: http://doi.acm.org/10.1145/988672.988679. (Cited on pages 43 and 115.)

[76] IBM 1997. IBM high performance compiler for Java, 1997. URL `http://www.alphaworks.ibm.com/formula/hpc`. (Cited on page 28.)

[77] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7): 1159–1176, Jul 2005. (Cited on pages 80 and 88.)

[78] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26. ACM, 2007. doi: http://doi.acm.org/10.1145/1238844.1238846. (Cited on page 155.)

[79] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for javascript. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1930–1937. ACM, 2009. doi: http://doi.acm.org/10.1145/1529282.1529711. (Cited on pages 43, 102, 103, 126 and 144.)

[80] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 238–255. Springer-Verlag, 2009. doi: http://dx.doi.org/10.1007/978-3-642-03237-0_17. (Cited on pages 42, 43, 102, 103, 126, 136, 143, 144, 145 and 152.)

[81] Graeme Johnson and Zoë Slattery. PHP: A language implementer's perspective. *International PHP Magazine*, pages 24–29, Dec 2006. (Cited on pages 76, 77 and 78.)

[82] Derek M. Jones. Forms of language specification: Examples from commonly used computer languages. ISO/IEC JTC1/SC22/OWG/N0121, February 2008. (Cited on page 70.)

[83] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 244–256. ACM, 1979. (Cited on page 22.)

[84] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 66–74. ACM, 1982. URL http://doi.acm.org/10.1145/582153.582161. (Cited on pages 21, 22 and 29.)

[85] Nenad Jovanovic. *Web Application Security*. PhD thesis, Technical University of Vienna, 2007. (Cited on pages 42, 102 and 145.)

[86] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 258–263. IEEE Computer Society, 2006. URL http://dx.doi.org/10.1109/SP.2006.29. (Cited on pages 42 and 145.)

[87] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. Technical report, Technology University of Vienna, 2006. (Cited on pages 62, 131 and 145.)

[88] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36. ACM, 2006. (Cited on pages 42, 102, 112, 116 and 123.)

[89] JRuby. JRuby. URL http://www.jruby.org. (Cited on pages 76, 77, 78 and 151.)

[90] Maria Jump and Benjamin Hardekopf. Pretenuring based on escape analysis. Technical Report TR-03-48, University of Texas at Austin, November 2003. (Cited on page 27.)

[91] Jython. Jython. URL http://www.jython.org. (Cited on pages 76, 77, 78 and 151.)

[92] J. Keith. *DOM scripting: web design with JavaScript and the Document Object Model*. Friends of Ed, 2005. (Cited on page 8.)

[93] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in ssa form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999. doi: http://doi.acm.org/10.1145/319301.319348. (Cited on pages 30 and 33.)

[94] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 107–120. ACM, 1998. doi: http://doi.acm.org/10.1145/268946.268956. (Cited on page 33.)

[95] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103. ACM, 1991. doi: http://doi.acm.org/10.1145/99583.99599. (Cited on pages 20, 22 and 24.)

[96] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 235–248. ACM, 1992. (Cited on pages 23, 31, 116 and 118.)

[97] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. *SIGPLAN Not.*, 28(6):56–67, 1993. ISSN 0362-1340. (Cited on page 21.)

[98] Christopher Lapkowski and Laurie J. Hendren. Extended SSA numbering: Introducing SSA properties to language with multi-level pointers. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 128–143. Springer-Verlag, 1998. (Cited on page 33.)

[99] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31. ACM, 1988. URL http://doi.acm.org/10.1145/53990.53993. (Cited on pages 21, 22, 29 and 129.)

[100] James Richard Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, EECS Department, University of California, Berkeley, May 1989. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/6161.html. (Cited on page 22.)

[101] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75. IEEE Computer Society, 2004. (Cited on page 88.)

[102] Alexandre Lenart, Christopher Sadler, and Sandeep K. S. Gupta. Ssa-based flow-sensitive type analysis: combining constant and type propagation. In

SAC '00: Proceedings of the 2000 ACM symposium on Applied computing, pages 813–817. ACM, 2000. doi: http://doi.acm.org/10.1145/338407.338570. (Cited on pages 30, 35 and 39.)

[103] Rasmus Lerdorf. Simple is hard. In *FrOSCon 2008*, August 2008. URL http://talks.php.net/show/froscon08. (Cited on page 2.)

[104] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282. ACM, 2002. URL http://doi.acm.org/10.1145/503272.503298. (Cited on page 39.)

[105] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. *SIGPLAN Not.*, 33(10):36–44, 1998. ISSN 0362-1340. URL http://doi.acm.org/10.1145/286942.286945. (Cited on page 28.)

[106] Nuno Lopes. Building a JIT compiler for PHP in 2 days, August 2008. URL http://llvm.org/devmtg/2008-08/. (Cited on page 88.)

[107] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. (Cited on page 37.)

[108] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 432–441. ACM Press, 2005. (Cited on page 43.)

[109] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-320-4. (Cited on pages 26, 47, 48, 132, 140 and 175.)

[110] H. Muhammad and R. Ierusalimschy. C APIs in extension and extensible languages. *Journal of Universal Computer Science*, 13(6):839–853, 2007. (Cited on pages 8, 10, 73 and 80.)

[111] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002. (Cited on page 52.)

[112] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007. ISSN 0362-1340. (Cited on pages 89 and 91.)

[113] Diego Novillo. A propagation engine for GCC. In *Proceedings of the GCC Developers' Summit*, pages 175–184, June 2005. (Cited on page 39.)

[114] Diego Novillo. Memory SSA–a unified approach for sparsely representing memory operations. In *Proceedings of the GCC Developers' Summit*, pages 97–110, July 2007. (Cited on pages 33 and 34.)

[115] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31:23–30, 1997. (Cited on page 7.)

[116] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 146–161. ACM, 1991. URL http://doi.acm.org/10.1145/117954.117965. (Cited on pages 14, 28, 36, 37, 41 and 144.)

[117] Young Gil Park and Benjamin Goldberg. Reference escape analysis: optimizing reference counting based on the lifetime of references. In *PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM, 1991. URL http://doi.acm.org/10.1145/115865.115883. (Cited on page 26.)

[118] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. *SIGPLAN Not.*, 27(7):116–127, 1992. ISSN 0362-1340. URL http://doi.acm.org/10.1145/143103.143125. (Cited on page 26.)

[119] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 67–78. ACM, 1995. doi: http://doi.acm.org/10.1145/207110.207117. (Cited on pages 30, 39, 61 and 149.)

[120] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007. ISSN 0164-0925. URL http://doi.acm.org/10.1145/1290520.1290524. (Cited on pages 29 and 158.)

[121] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 0-262-16209-1. (Cited on pages 11, 12 and 109.)

[122] Anthony Pioli, Michael Burke, and Michael Hind. Conditional pointer aliasing and constant propagation. Master's thesis, SUNY, New Paltz, 1999. (Cited on pages 39, 61, 121, 144 and 145.)

[123] Quercus. Quercus: PHP in Java, 2008. URL http://www.caucho.com/products/quercus/. (Cited on pages 75, 77 and 78.)

[124] Norman Ramsey, Joao Dias, and Simon P. Jones. Hoopl: Dataflow optimization made simple. 2009. (Cited on page 39.)

[125] Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM Press, 2004. (Cited on pages 16 and 88.)

[126] Roadsend. Roadsend PHP. URL http://www.roadsend.com/. (Cited on pages 75, 77 and 78.)

[127] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988. URL http://doi.acm.org/10.1145/73560.73562. (Cited on page 30.)

[128] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 47–56. ACM, 2000. URL http://doi.acm.org/10.1145/349299.349310. (Cited on page 29.)

[129] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–293. ACM, 1988. URL http://doi.acm.org/10.1145/73560.73585. (Cited on pages 22 and 26.)

[130] B. Ryder. Dimensions of precision in reference analysis of object-oriented languages, 2003. (Cited on page 23.)

[131] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–31. ACM, 1996. URL http://doi.acm.org/10.1145/237721.237725. (Cited on page 29.)

[132] Michael Salib. Starkiller: A static type inferencer and compiler for Python. Master's thesis, Massachusetts Institute of Technology, 2004. (Cited on pages 41, 77, 78, 91 and 144.)

[133] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search presentation. In *Velocity 2009: Web Performance and Operations Conference*. O'Reilly, June 2009. URL http://en.oreilly.com/velocity2009/public/schedule/detail/8523. (Cited on page 2.)

[134] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1997. URL http://doi.acm.org/10.1145/263699.263703. (Cited on page 25.)

[135] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981. (Cited on pages 23 and 124.)

[136] Olin Grigsby Shivers. *Control-flow analysis of higher-order languages of taming lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991. (Cited on page 145.)

[137] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, 1995. URL http://doi.acm.org/10.1145/199448.199464. (Cited on page 31.)

[138] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996. URL http://doi.acm.org/10.1145/237721.237727. (Cited on pages 24, 25 and 26.)

[139] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–95. ACM, 2000. URL http://doi.acm.org/10.1145/325694.325706. (Cited on page 29.)

[140] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. An empirical study of method in-lining for a Java just-in-time compiler. In *Proceedings of the*

2nd Java Virtual Machine Research and Technology Symposium, pages 91–104. USENIX Association, 2002. (Cited on pages 15 and 88.)

[141] The PHP Documentation Group. *PHP Manual*, 1997-2009. URL http://www.php.net/manual/. (Cited on page 110.)

[142] The PHP Group. Zend benchmark, 2007. URL http://cvs.php.net/viewvc.cgi/ZendEngine2/bench.php?view=co. (Cited on pages 91 and 138.)

[143] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293. ACM, 2000. URL http://doi.acm.org/10.1145/353171.353190. (Cited on page 35.)

[144] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., 2007. ISBN 012088478X, 9780120884780. (Cited on pages 59 and 132.)

[145] Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. Copy-on-write in the PHP language. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 200–212. ACM, 2009. (Cited on pages 86 and 105.)

[146] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM, 1987. (Cited on page 14.)

[147] Kenneth Walker and Ralph E. Griswold. An optimizing compiler for the Icon programming language. *Softw. Pract. Exper.*, 22(8):637–657, 1992. ISSN 0038-0644. URL http://dx.doi.org/10.1002/spe.4380220803. (Cited on page 78.)

[148] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41. ACM Press, 2007. (Cited on pages 43, 61, 99, 102, 114, 117, 127, 136, 144, 149 and 152.)

[149] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international sympo-*

sium on Software testing and analysis, pages 249–260. ACM, 2008. URL http://doi.acm.org/10.1145/1390630.1390661. (Cited on page 159.)

[150] Ben Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. Software Eng.*, 1(3):270–285, 1975. (Cited on pages 38 and 39.)

[151] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. (Cited on pages 30, 36, 38 and 64.)

[152] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144. ACM, 2004. URL http://doi.acm.org/10.1145/996841.996859. (Cited on pages 29 and 158.)

[153] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206. ACM Press, 1999. (Cited on pages 26 and 27.)

[154] Andy Wharmby. Understanding PHP opcodes, September 2006. URL http://www.zapt.info/PHPOpcodes_Sep2008.odp. (Cited on page 90.)

[155] Kevin Williams. *Data and Type Optimizations in Virtual Machine Interpreters–in preparation*. PhD thesis, Trinity College Dublin, 2009. (Cited on pages 17, 18 and 41.)

[156] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium,*, pages 179–192, July 2006. (Cited on pages 43, 102, 115, 119 and 126.)

[157] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93, 2007. (Cited on page 88.)