Applying Optimistic Optimization Techniques to Dynamic Programming languages

Maxime Chevalier-Boisvert

September 2, 2011

Contents

1 Introduction **2**
1.1 Compilers and Virtual Machines 2
1.2 Dynamic Programming Languages 4
1.3 Optimization Challenges 6
1.4 Methodology 7
1.5 The JavaScript Language 8
1.6 About this Report 9

2 Related Work **10**
2.1 Static Program Analyses 10
2.2 Program Transformations 12
2.3 Adaptive Optimizations 13
2.4 Behavior of Dynamic Languages 14
2.5 Optimizing Dynamic Languages 15
2.6 Trace Compilers 16
2.7 Optimistic Optimization 17
2.8 Garbage Collection 18
2.9 Metacircular Systems 20

3 Objectives **22**
3.1 Optimistic Optimization 22
3.1.1 Powerful Optimizations 22
3.1.2 Code Example 23
3.1.3 Compilation Model 26
3.1.4 Performance Analysis 27
3.1.5 Potential Limitations 29
3.2 Success Criteria 30
3.3 Metacircular Implementation 32
3.4 Long Term Perspectives 33

4 Conclusion **34**
4.1 Current State of the Project 34
4.2 Thesis Time Table 35
4.3 Future Programming Languages 38

Chapter 1

Introduction

Dynamic programming languages are a specific kind of programming language that is said to be easier to use, to improve productivity and to reduce development time. Unfortunately, so far, programs written in dynamic languages have suffered from an important performance weakness when compared to traditional static programming languages such as C and Java. This deficiency is largely because dynamic languages are harder to optimize. For my thesis, I aim to demonstrate that more efficient implementations of dynamic programming languages can be obtained by using a new design which allows deeper code optimizations.

1.1 Compilers and Virtual Machines

The first general-purpose electronic computer, called ENIAC (for Electronic Numerical Integrator And Computer), was completed in 1946 and weighed nearly 30 tons. To program this computer, programs were first devised and verified on paper, a process which could take several weeks. The programs were then painstakingly entered into the computer manually using large switchboards and patch cables. The switch positions were meant to directly represent hardware instructions. There was no way to store programs into any kind of permanent storage. Computers were soon improved so that they could be programmed using reusable storage such as punched tapes and perforated paper cards.

This was nevertheless still a crude low-level way to program them. One important innovation in the history of computers was the invention of assembly languages. These are textual representations of machine instructions which use short mnemonic names for the instructions. Assembly languages allowed programmers to edit their programs textually using a keyboard and a terminal. Once the editing was complete, a program could be automatically translated to machine instructions by a simple compiler. Once compiled, the program could be executed as many times as desired.

Assembly languages greatly facilitated the programming of computers, but

they were still very verbose and tedious to use. Another issue is that these languages were specific to the processor and operating system on which the program was to run. As such, an assembly program that ran on one computer might not even run on the next generation of computers from the same vendor. A solution to this problem came in the forms of high-level programming languages. The first such language, FORTRAN, appeared in 1954. High-level languages allow programmers to write functions and declare variables without the need to concern themselves with the specific instruction set to which the programs will be compiled.

To make high-level languages possible, more sophisticated compilers were needed, which could map high-level language constructs such as loops, functions and arithmetic operations down to machine instructions. The first compilers were *ahead of time* compilers. That is, they compiled programs all at once, before any of their code was executed. Such compilers needed to integrate several optimization algorithms in order for the machine code they generated to be efficient, and competitive with hand-written assembly code. In the case of FORTRAN, these optimizations were largely helped by the inclusion of type annotations in programs, which told the compiler what kind of data any given variable would store during the program's execution.

Many of the first programming languages resembled FORTRAN in their use of type annotations and ahead of time compilers. These were a significant improvement upon assembly languages, but some programmers still found important downsides with such languages. One such downside is that languages like FORTRAN, while more expressive than assembly, were still restricted in many ways. For one, the type annotations made the code quite verbose. These type annotations also corresponded to restrictions on what the programs could do at execution time. If one wanted to make a function to add the contents of a list of integers in FORTRAN, for example, one could not reuse this same function to sum a list of real numbers. Even though both operations are almost the same from a conceptual standpoint, in FORTRAN, two similar but distinct functions would need to be written.

The LISP programming language was invented by John McCarthy in 1958. This language was quite different from FORTRAN in that it did not require the programmer to write type annotations, making code more reusable. Its main feature, however, is that it used the same representation (lists) for both code and data. This would allow programs to generate new code dynamically at *run time* (during their execution).

The first implementation of LISP was not realized using an ahead of time compiler, but rather by executing the code inside a program known as a *Virtual Machine* (VM), which is a software implementation of a computing machine designed more or less specifically for the source language. A VM essentially simulates the execution of the program on the machine. VMs are traditionally implemented using an interpreter. Such an implementation is not optimal in terms of performance, but made it easier to implement the runtime support necessary to implement the dynamic semantics of LISP. This runtime support included garbage collection and the facilities needed to parse and execute new

LISP code at run time.

Several modern language implementations rely on VMs. This is in part due to their runtime support needs, and also because of the portability advantage offered by such an approach. Namely, when using an ahead of time compiler, one must compile a different version of a program for each architecture on which it needs to run. Using a VM, however, the programmer does not need to worry about this. In the case of Java, for example, programs are compiled down to an abstract instruction set known as Java bytecode. The compiled programs can then be executed by the Java VM on any platform for which there is a Java VM. The programmer does not need to compile multiple versions of programs. In the case of the Python language, a VM is also supplied, which compiles programs from Python source code to bytecode when the programs are launched.

The first Java VMs, just as the first LISP and Smalltalk VMs, used interpreters to execute programs. Unfortunately, the use of an interpreter often results in performance one to three orders of magnitude slower than compiling programs directly to machine instructions, because of the added simulation overhead. A popular alternative, used in more modern VM implementations, is to integrate what is known as a *Just-In-Time* (JIT) compiler in the VM, so that the code executed by the VM can be compiled down to native machine code just before it is run. This can yield much better performance than an interpreter, but it is usually more complex to implement than a static ahead of time compiler.

JIT compilers have an interesting advantage over ahead of time compilers. Ahead of time can only examine a program's source code at *compile time* (while it is being compiled, before any code is run). This sometimes makes it difficult for them to predict the behavior of programs at run time (while they execute). JIT compilers, on the other hand, can delay compilation and examine the behavior of programs at run time. This can allow them to tailor optimizations to the specific behavior of the program being executed.

1.2 Dynamic Programming Languages

Programming languages can be loosely divided into two categories: static and dynamic languages. Static languages are the most well known. This family includes languages such as C, C++, FORTRAN, Java, ML, etc. These are usually compiled ahead of time and *statically typed*. That is, each variable in a static language has a fixed type; it can only store one type of value (e.g.: strings, booleans, integers). The types of variables are either specified using explicit annotations written in the program's source code (as in C, Java), or by using a constraint-based type inference system which tries to resolve all variable types simultaneously (as in ML).

One of the main difference between static and dynamic languages is one related to the concept of binding time. In static languages, all variables, operator and function declarations must be resolved at compilation time (they are *statically bound*), whereas dynamic languages tend to delay the resolution of

global variables and functions until run time (they are *late bound*). Dynamic languages, as the name implies, have more dynamic characteristics. The degree of dynamism is not the same for all dynamic languages, but they often sport a similar set of features. I define dynamic languages as languages that include at least the following four features:

1. Dynamic typing: variables can change type at any point during a program's execution. Variable types are not constrained at compile time, rather, the type-correctness of the program is verified at run time.
2. Dynamic dispatch: function call semantics imply a dynamic method lookup (method names are late bound).
3. Dynamic loading: it is possible to generate, load and execute arbitrary new code using an `eval` function.
4. First-class functions: it is possible to obtain a reference to a function and store it as a value. It is also possible to redefine global functions at any time.

Such languages include Scheme, JavaScript, Smalltalk, Python, Ruby, PHP, Perl and MATLAB.

Dynamic typing allows writing potentially more compact and expressive code, since the types of variables are not fixed ahead of time and any function can possibly be applied to an infinity of combinations of argument types. *Dynamic dispatch* means that a function needs to be defined only when it is called, and can be redefined at any point while the program executes.

The `eval` function typically takes a run time generated representation (such as string) of arbitrary source code and executes it immediately. This potentially allows for a program to dynamically generate arbitrary new code and inject it within itself at any point during its execution. Using `eval`, new variables can be defined and functions can be redefined at any time. This is a powerful tool, as it makes it possible to write code that generates new code. It is probably the most expressive form of metaprogramming. The `eval` function is often used in practice to implement serialization and plugin systems.

Other characteristics common to almost all dynamic languages include the presence of a garbage collector, as well as the inclusion of higher-level data structures, such as lists and dictionaries, among the native types supported by the language. This usually implies the presence of a syntax to generate instances of these data structures as literals in the source code. This kind of feature largely aim to simplify and accelerate the writing of code in these languages.

Obviously, dynamic languages are not without disadvantages. Static type verification allows discovering some programming bugs which may, in a dynamic language, only be discovered when they manifest themselves at execution time. Another potential weakness is that it is much more difficult to efficiently optimize code written in a dynamic language. The main reason being that the syntax of dynamic languages is much less explicit than that of statically typed

languages, particularly in terms of type information, and thus carries significantly less semantic information about the program.

Despite these disadvantages, the popularity of dynamic languages has been increasing rapidly in the last two decades. They have acquired the reputation of being much easier to learn and use than static languages, and most agree that it is significantly faster to write a program in a dynamic language than in a static one [1]. Dynamic languages power the modern dynamic web, and thus, their popularity should be expected to grow even more in the foreseeable future, as companies such as Google and Microsoft seek to move more and more software to web-based platforms.

An important problem, however, is that, to this day, dynamic languages are still not able to rival the performance of static languages. They are often one to two orders of magnitude slower. In a world where we worry about energy savings and where future applications will become more and more complex, while the evolution of computer hardware seems to slow down, it seems possible that this performance disadvantage may become a barrier. In light of this, I aim to find ways to improve the performance of dynamic languages, making them even more competitive with their static counterparts.

1.3 Optimization Challenges

In order to get the best performance from dynamic languages, one must be able to compile them down to machine code. However, the simplest way to implement a dynamic language is often to implement the language in an interpreter. This is in part because dynamic languages need a significant amount of runtime support. Runtime support includes the garbage collector, but also dynamic code loading, which means that the executing program could trigger the execution of arbitrary code. The capability to execute arbitrary code implies that, if we implement a compiler for a dynamic language, the compiled programs must potentially have access to the compiler itself, which complicates the implementation of such a compiler.

Another factor which makes the static compilation of dynamic languages problematic is that the behavior of programs to be compiled is often hard to predict. In JavaScript, for example, the `+` operator does not only serve to add numbers. It can also serve to concatenate strings, or to concatenate strings and numbers. This implies that at each use of this operator in a program, there are several possible cases to handle. It is unfortunately difficult to statically know which specific ones apply ahead of time.

Some characteristics of dynamic languages are simply not well-suited to static compilation. In JavaScript, for example, it is possible to define new variables dynamically inside of function bodies using the `eval` function. This makes it so programs, at run time, can access variables and functions which did not initially exist in the program that was originally compiled.

Traditional compiler optimization techniques have been developed in the context of static language compilation. These techniques are said to be con-

servative, meaning that the compiler assumes there is no runtime support for reoptimization, and must prove at compile time that any optimization to be applied will preserve the program semantics for all inputs. In this context, static languages are easier to optimize because the types of variables are known at compile time. The C language possesses no dynamic typing or operator overloading. This generally allows implementing operations such as addition and comparison of numerical variables directly using only one machine instruction.

Furthermore, in the case of static language compilers, the source code to be compiled is generally fixed and entirely accessible to the compiler at compilation time. This allows applying relatively simple analyses on the programs to determine the applicability of optimizations (for example, pointer aliasing analyses). The compiler can count on the fact that new code cannot be introduced dynamically during execution and invalidate optimization decisions.

Unfortunately, the techniques that are well-suited to static languages are not sufficient to make dynamic languages competitive. In the context where code can be loaded dynamically at any time, static analyses are simply no longer able to guarantee the validity of most of their predictions, since new code could redefine the type or the value of any global variable (and even local variables in the case of `eval`), including globally defined functions.

Simple optimizations, such as inlining, also become less directly applicable. How can one inline a function, if there is no static guarantee as to which function is being called? Dynamic typing makes optimization problematic, since variables can suddenly change type during execution, and it is difficult to prove that this will not happen in a large proportion of cases. The optimization of dynamic languages is thus a major challenge. More advanced techniques combining partial evaluation and dynamic profiling with traditional optimization techniques are necessary to obtain significant performance gains.

In theory, a Just-In-Time (JIT) compiler can generate more efficient code than a static compiler, since the former can observe the state of the compiled program as it is executing. Many dynamic language compilers are beginning to take advantage of this using simple techniques, such as trace compilation. However, since there remains a major difference (often an order of magnitude or more) between the performance of static and dynamic languages, I believe there remains a large potential to be exploited.

1.4 Methodology

I find dynamic languages interesting. I believe that it is possible to develop new optimization techniques that are better suited to such languages, and will help make them more competitive with static languages. As explained in the rest of this proposal, I will be studying optimistic optimizations, a new class of optimization techniques I believe to be better adapted to dynamic languages than what is currently used in production compilers.

This research work is experimental. To validate the potential of these optimizations, I will need to design new analyses and program transformations,

and implement them in an actual compiler. By testing these optimizations and observing the resulting performance over a suite of benchmark programs, I will gain the necessary feedback to understand the potential flaws in my approach. This improved understanding will then allow me to iteratively improve the optimizations devised.

Because of time constraints, I have chosen to focus my study on one programming language, namely JavaScript. It is fairly representative of dynamic languages in terms of its features and capabilities. It is also probably fair to say that it is one of the more dynamic languages among this category. The language, while nontrivial, is simple enough that it is realistic to think that I can account for most if not all of its features in my research work. JavaScript is also possibly the most widely used dynamic language at this time, being that it is used to do client-side programming of webpages in web browsers, making its optimization very relevant.

The compiler I will be implementing my optimizations into is called Tachyon. It is an experimental compiler for the JavaScript programming language which has been created at the Laboratoire de Traitement Parallèle at the Université de Montréal. I have been working on this compiler myself, in conjunction with Professors Marc Feeley and Bruno Dufour, as well as M.Sc. student Erick Lavoie. It is fairly complete at this stage, and supports most of the ECMAScript 5 specification, with the eventual goal of supporting it in its entirety.

1.5 The JavaScript Language

Many make the mistake of believing that JavaScript is a dialect of Java. However, these languages are in fact quite different. Although the syntax of JavaScript superficially resembles that of Java, and JavaScript is also an imperative language, JavaScript has much more in common with Python and Smalltalk. It is a dynamic language with a strong functional component and an object system based on the concept of prototyping, similar to that of SELF [2].

JavaScript could be said to be object-oriented, but it does not comprise a system to define classes statically as in C++ or Java. It is in fact possible to add and remove fields (called properties in JavaScript) on an object at any time (see fig. 1.1). The concept of prototype-based inheritance implies that an object can have a parent on which the value of a property may be found if it is not defined on the object itself. There is a chain of prototypes, in fact, since the parent object can also have a parent of its own.

An interesting aspect of JavaScript is that it has higher order functions, and thus allows creating closures. It is also possible to write nested functions which may capture local variables of all outer functions. This is practical to express some operations in a more concise manner. JavaScript offers, for example, `map`, `forEach` and `filter` functions, which allow manipulating elements of an array of data using closures (see fig. 1.2), in a similar manner to what other functional languages like Scheme and ML allow.

JavaScript also possesses reflective functionalities. It is possible, for example,

```
d8> var obj = { x:1, y:'foo' }
d8> obj.z = 3
3
d8> for (k in obj) print(k);
x
y
z
d8> delete obj.z
true
d8> print(obj.z)
undefined
```

Figure 1.1: Adding, enumerating and deleting properties of an object in the Google V8 interactive shell.

```
d8> function add(x) { return function (y) { return y + x; } }
d8> var arr1 = [11,22,33];
d8> var arr2 = ['book','bag','apple'];
d8> print(arr1.map(add(100)));
111,122,133
d8> print(arr2.map(add('s')));
book,bag,apple
```

Figure 1.2: Using `map` to add numbers and concatenate strings.

to enumerate the names of properties defined on an object. It is also possible to access an object's property using its name, in the form of a character string. This allows using objects as dictionaries. These are, however, indexable using only strings.

1.6 About this Report

This report is divided into four chapters. Chapter 2 presents related work in the field of compiler optimization. This includes traditional analysis and optimization techniques designed for static and dynamic languages as well as recent developments in dynamic language optimization. Background about dynamic language runtimes is also presented.

Chapter 3 delves into optimistic optimization techniques, the novel ideas I want to explore in my research, and the kinds of program transformations a compiler using optimistic optimizations ought to be able to accomplish. This chapter also contains an analysis of the anticipated performance implications of the optimistic optimizations I intend to test. Finally, chapter 4 concludes with information about the current state of the Tachyon compiler and a time table for my thesis research work.

Chapter 2

Related Work

2.1 Static Program Analyses

The vast majority of research on compiler optimization has been done in the context of static compilation. Typically, in this compilation model:

- The compiler has access to all the source code of the module being compiled.
- Dynamic loading of new code is not possible, or otherwise, dynamic loading is done through a restricted interface (e.g.: loading of entire modules in separate namespaces).
- Compilation time is not very important, which allows the compiler to take much longer to analyze and optimize code.
- Typing information is determined statically, either by type annotations (as in C and Java), or by type inference (as in ML and Haskell).

In the context of ahead of time compilation, before optimizations are performed, the compiler may run analyses on the program being compiled. These serve to gather additional information about its possible run time semantics. More specifically, analyses are often used to prove specific properties about the program, so that the compiler can know which analyses are safe to apply. There are many well-known compiler analyses. Some of the most common ones include:

- Control-flow analysis: aims to determine the potential receivers of function calls to build a control-flow graph of the program. This can enable inlining, dead code eliminations.
- Alias analysis: determines which references or pointers may alias so that operations can be safely reordered.

- Type inference: infers the type of local and global variables. This can be done with varying degrees of precision. The resulting information can be used to specialize the program.
- Escape analysis: aims to determine when the scope of a given object is restricted (e.g.: an object is only live within a given method invocation).

Note that there are many different ways in which each of these analyses can be implemented, each with different speed and accuracy tradeoffs. One of the most important analysis techniques developed in the context of static optimization is that of abstract interpretation [3]. This technique essentially aims to simulate the execution of an entire program, or part of a program (such as a function), over abstract symbolic values. These abstract values can represent a property about which we wish to obtain information, such as, for example, the numerical intervals a variable's value can range over. Abstract interpretation also yields itself fairly well to certain type analyses [4, 5, 6].

A problem in the case of dynamic languages is that, as previously explained, the information available directly from the source code is relatively poor and little is directly known about the types of variables. If static analyses are attempted, many unknown values end up propagating through the said analyses, and the analysis results are often poor as well. Another problem is that, in the context of JIT compilation, these analyses are often considered too expensive in terms of execution time. The time necessary to run the analyses must be added to the execution time of the program being analyzed, which introduces a compromise between the potential performance gain linked to an analysis and its run time cost. Effectively compomising between these two quantities requires a cost-benefit analysis [7].

Despite the difficulties associated with the use of static analyses with dynamic languages, many researchers are still examining this topic. Unfortunately, this is often done with the idea that the dynamic behavior of the analyzed programs can be ignored. The assumption is made that it is possible to access the entirety of the code to be analyzed. The presence of the (in)famous `eval`, for example, is often completely ignored, and left as an exercise to the reader. This essentially comes down to treating dynamic languages as if they were purely static.

In this context, type inference analyses have been applied to JavaScript. Zhao has developed a set of constraints which allow inferring the type of objects and properties of a JavaScript program [8] similarly to the type inference algorithm of ML [9]. This is costly and problematic in the context of dynamic code loading, because strong constraints on types are poorly adapted to this situation, but the main objective of this research was to detect potential execution errors offline. The same kind of approach has also been applied to the Ruby language [10].

In the same lineage, Guha et al. have developed a simplified language in the style of lambda calculus, to which they translate JavaScript [11]. This language has a well-defined formal semantic, is simpler to analyze, and is well-suited to proof techniques. The big limitation, however is that this translation

ignores `eval` and dynamic code loading. The lambda-JS calculus invented by the authors focuses on the aspects of JavaScript said to be "essential", which means its object model. Jang and Choe have taken a similar approach to implement an alias analysis for JavaScript. This analysis is based on a translation of JavaScript to a simplified language called SimpleScript, and the generation of a set of constraints to be solved [12].

A different approach to type analysis of JavaScript programs has been put forth by Jensen et al. [6]. It is based on the abstract interpretation of programs. Their analysis is purely static, and thus suffers from the same limitations as the previously enumerated approaches. However, I believe that Jensen's approach is possibly better suited to dynamic languages, because abstract interpretation is less fragile, in a dynamic context, than an inference system based on absolute typing constraints. Abstract interpretation can be performed incrementally, for instance.

2.2 Program Transformations

Compilers can transform programs with the goal of improving their performance. One of the most important and well-known such transformation is known as function inlining. This transformation copies and inserts the code of a function at points in the program that call it (its call sites), substituting variable names as appropriate. Inlining can improve performance by eliminating function call overhead and by allowing the compiler to specialize the inlined code based on the calling context. It is particularly beneficial for languages which encourage many calls to small functions or methods, as is the case in OOP languages such as Java.

One must carefully choose which functions to inline so as to avoid increasing code size too much, and incurring an instruction cache penalty. As such, the functions that are inlined are usually small functions, frequently called functions, or functions with one or few associated call sites. It is difficult to perform efficient inlining in dynamic languages [13], however, because the compiler may not have guarantees as to which method is called by a given call site, and whether or not this method may be redefined in the future. This issue is usually circumvented by inserting guards before inlined code, which test that the method being called is indeed the one that was inlined. This is a form of speculative optimization.

Other common compiler optimizations include constant propagation, which replaces variables with constant values by their value, common subexpression elimination, which eliminates duplicate expressions in programs, and loop invariant hoisting, which moves invariant code outside of loops whenever possible. Loop invariant hoisting is particularly important for dynamic languages because it can allow the elimination of costly bound checks [14] inside loops, for example.

An interesting optimization developed in the context of static languages is procedure cloning [15]. This technique involves duplicating the code of a function (without inlining it) and specializing it for multiple sets of call sites. It

is particularly attractive for dynamic languages, because the types of parameters to a function can be relatively stable at specific call sites. It is also possible to specialize functions on demand by intercepting parameters that are passed to them [5].

2.3 Adaptive Optimizations

Adaptive optimizations are optimization techniques that are meant to adapt to a specific program, usually by profiling it or examining its state as it is running. Lee & Leone have proposed a technique called deferred compilation [16]. This technique involves the generation of code at run time. More specifically, some methods are compiled and optimized based on the value of certain arguments, which are only available at run time, using a form of partial evaluation to generate machine code. This could be said to represent a precursor to the JIT compilers of today.

In modern JIT compilers, one of the simplest and more common forms of adaptive optimization is to have multiple optimization levels for compiling methods [7]. Often, a baseline compiler is used to generate code rapidly for the first compilation of any method, allowing the JIT compiled programs to start faster. This baseline compiler does not generate very efficient code, but the “hot” methods (which consume the most execution time) can be detected and recompiled at higher optimization levels. Which methods get compiled at which optimization level usually depends on some cost-benefit model. Techniques such as on-stack replacement [7, 17] can be used to replace the code for hot methods as they are running.

A more advanced form of adaptive optimization involves profiling the values on which a program operates in order to specialize it. Some of the first compilers to do this were static compilers. Muth et al. have used such a technique to optimize C code. In their implementation, the code is first profiled offline, and then optimized based on common expression values. Their optimizations involve adding new run time checks to detect specific frequent values. They have developed an analytical model to try and predict where specialization will be beneficial [18]. A similar idea has been applied to SELF. Hölzle & Agesen have used a technique called type feedback to monitor the types of possible receiver classes at call sites during previous runs. They claim that this technique performs just as well, and sometimes better, than static type inference [19].

Such techniques have since been integrated into several JIT compilers. Whaley implemented a Java VM which is able to specialize basic block ordering based on profiling data [20] obtained at run time. More recently, there has been an interest in applying type feedback to dynamic languages at run time. This seems like a sensible idea, since dynamic languages usually run in a VM, and it is notoriously difficult to determine the types of values in these languages using solely type inference. Furr et al. have applied type feedback to translate Ruby code into a type-annotated variant [21]. Williams et al. have applied it to Lua, using an interpreter to profile variable types, and then applying a dataflow-based type

inference technique to the result [22]. They have shown significant performance gains over the baseline Lua interpreter.

Another form of run time optimization is on-demand specialization. This technique, pioneered by the Psyco Python JIT [23], aims to specialize code while it is executing. In Psyco, methods can capture the types of values they operate on, and invoke the JIT compiler to be further specialized. In my own M.Sc. thesis, I have proposed a similar idea which involves capturing the types of method arguments at method invocations sites, and lazily compiling specialized versions of the said methods based on argument type information, which is used to determine local variable types using type inference [5]. Logozzo and Venter from Microsoft Research have proposed a similar technique [4].

Another novel idea in the realm of adaptive optimizations is that if an optimizer can observe the state of a program as it is running in order to efficiently optimize it, it might be beneficial to do this more than once. More specifically, it might be useful to reoptimize the program at different times during its execution, especially if the program enters different phases during which it operates on different kinds of data. This idea is explored in a case study by Kistler & Franz [24]. They have demonstrated speedups of over 120% in some cases.

2.4 Behavior of Dynamic Languages

In order to efficiently optimize dynamic languages, it is essential to study their run time behavior, so as to know the nature of the issues involved. To this day, it seems that the assumptions made about the behavior of programs written in dynamic languages are based on the hypothesis that these are never maximally using the dynamic capabilities of the language, and behave mostly like programs written in static languages. The "common sense" assumptions made about languages like C++ and Java are thus assumed to remain true for dynamic languages.

Recently, a few empirical studies about the behavior of Python [25] and JavaScript [26, 27] programs have appeared. These studies have demonstrated that, as was already assumed, the large majority of call sites in dynamic programs are in fact monomorphic (always call the same function), the majority of JavaScript programs do not use property deletion at all, and that the use of dynamic code loading, for example using `eval`, while sometimes non-trivial, remains relatively rare.

However, these studies have exposed the fact that, particularly in JavaScript, the run time behavior of the most used benchmark programs is in fact very different from that of popular web applications. This could be a problem because the most widely used compilers in the industry base their design choices on the performance obtained on the said benchmark programs. Hence, it is justified to question whether these compilers are being over-optimized for the wrong kinds of tasks.

Since few studies have been made thus far as to the behavior of dynamic programs, many questions remain unanswered to this day. To our knowledge,

there is no published empirical data, for example, about the stability of types of global variables or function parameters and return values. Our own small-scale experiments lead us to believe that variables in JavaScript programs are usually fairly type-stable, but more research is necessary on this topic in order to guide research on dynamic language optimization.

2.5 Optimizing Dynamic Languages

Dynamic languages have recently known significant gains in popularity, but they are not new. In fact, among the first dynamic languages are comprised many well-known LISP and BASIC dialects, which are traditionally dynamically typed. As such, dynamic languages predate C, C++ and Java by decades. Their performance, however, has always been an issue and originally, the general opinion was that the only way to make dynamic languages fast was to conceive computers specifically optimized for them (such as the MIT LISP Machines).

This opinion began to change near the end of the 1980s with the arrival of SELF [2], a language inspired from Smalltalk which, through effective optimizations, offered a competitive level of performance on general-purpose processors. The researchers working on the SELF language have put forth many dynamic language optimization techniques, including static type analyses, more efficient data representations for dynamically-typed values, as well as run time code specialization techniques, such as code patching.

In dynamic language implementations, it is essential to keep track of the type of values used. Many techniques have been developed to efficiently represent dynamic typing information [28]. One of the most practical such techniques is the use of tag bits in dynamically typed values. This technique uses a few of the lower bits of values to encode type information. The information can then be rapidly accessed using bitwise operators. One bit can be used, for example, to distinguish between pointers and integer values.

An important problem in object-oriented dynamic languages is the memory representation of objects. In most of these languages, it is possible to add and remove properties on objects dynamically. This implies that an object behaves somewhat like a hash map, where each property name is associated to a value. Unfortunately, while hash maps are easy to implement, they are inefficient object representations in terms of space usage and execution time.

An object representation developed for SELF assigns a different identifier to each set of properties an object can have at execution time. Each set of properties is associated to what is called a “map”. This map essentially describes a memory layout for a given kind of object, having the properties contained in the corresponding set. Each object thus has an associated “map”, and dynamically changes representation when properties are added to or removed from the object. This allows representing objects more directly in memory, in the same way as a C `struct`, without needing to encode the set of properties present on each object directly on the objects themselves.

This representation is well-suited to many optimization techniques. It can

be efficiently combined with code patching to cache field offsets. It allows faster access to properties and more compact objects. This is why variants of this representation are used in the most competitive dynamic language implementations of today, such as Google's V8, Mozilla's SpiderMonkey and PyPy.

Code patching is often used in dynamic language compilers. As the name implies, the idea is that the compiler is able to rewrite portions of executable code during execution. This technique is used, among other things, in the SELF VM for example, to implement local caches at call sites. These caches are used to try and predict the type of the receiving object of method calls [29]. This allows reducing the number of dynamic method property lookups. Code patching is also used in Google V8 to try and predict the type of objects for property offset lookups. Generally, code patching is effective for applications where it is needed to locally predict frequent run time behaviors, but it may be necessary to change the prediction being made as the program runs.

2.6 Trace Compilers

A recent development in the world of dynamic languages is the use of trace compilers. These have been originally developed with the goal of optimizing precompiled machine code [30]. It is difficult to make a complete decompilation of already compiled programs. Trace compilers sidestep this problem by emulating the execution of the compiled program in order to produce a trace of the most often executed code. A trace is simply a linear sequence of frequently executed code. These can be optimized separately from the rest of the program using simple algorithms that operate on long instruction sequences.

The creators of HotpathVM have demonstrated that, in the context of Java, trace compilation can be a competitive JIT compilation model [31]. HotpathVM uses an interpreter to record traces of the hottest (most executed) loops in a Java program. These are then JIT compiled to machine code. This technique has several advantages:

1. The compiler can be kept simple. It only needs to compile linear sequences of code. It does not need to support all language constructs as unsupported ones can still be interpreted.

2. The resulting compiler can be very vast. Only the hottest traces are compiled, which may be a small fraction of the entire program.

3. Traces are well-suited to optimization. They are long linear sequences of code that are infrequently exited, similarly to extended basic blocks.

More recently, trace compilers have been applied to a few dynamic languages, such as Lua, Python and JavaScript. A trace compiler is used in the Firefox web browser, and has permitted a great improvement of its JavaScript execution performance [32]. Other researchers have also applied trace compilers to JavaScript, and obtained performance gains in excess of an order of magnitude compared to an interpreter [33].

Since the arrival of trace compilers, it has been shown that it is possible to integrate low-cost optimizations to this compilation model, such as type specialization [32], elimination of superfluous allocations [34], as well as analyses used to eliminate redundant overflow checks [35]. It has also been demonstrated that the total execution time of a program can be reduced by compiling traces in a concurrent manner [36]. Trace compilation is currently very popular in the field of dynamic language optimization, and there is little doubt that more improvements to this technique will be discovered.

2.7 Optimistic Optimization

As explained in section 1.3, applying traditional static optimizations to dynamic languages directly is problematic. This is mostly due to the fact that programs written in dynamic languages are difficult to analyze statically. It is difficult to prove conservatively that a property necessary for a given optimization will always be true for all possible future states of a dynamic program.

The potential dynamism of this kind of program makes those guarantees difficult to obtain. However, in practice, dynamic programs are not the product of randomness, and do not exploit the dynamism of their implementation language in an uncontrolled manner. In most cases, we can expect that these pograms, by their algorithmic nature, will execute repeated computations on similar data sets, and thus will behave in a relatively stable and predictable manner.

The fundamental idea behind optimistic optimizations is that many optimizations may be applicable to a program in its current state of execution. Even if we cannot prove that these optimizations will be applicable to all possible future states of the said program, we can suppose, in an optimistic manner, that it will be the case. Applying optimizations in this way is safe, so long as the optimizations can be deactivated when the assumptions on which they are based are found to be false. Optimistic optimization lifts the burden of conservative proof and instead relies on runtime support for reoptimization to guarantee the correctness of the optimized code.

Inline caching and trace compilation can be seen as limited forms of optimistic optimization, because these optimizations generate optimistic code based on local properties measured at run time. The generated code assumes that the behavior of the program will remain stable for a certain amount of time, and can generally be regenerated if the behavior of the program changes. The main disadvantage of this kind of technique, however, is that it is based on a division between a fast (common) execution path and a safe (slow) execution path. Dynamic checks have to be frequently executed to verify that the fast execution path is still applicable.

The concept of optimistic optimization has also been explored in the context of Java. This language is statically typed and does not have an `eval` function, which makes it easier to analyze statically than JavaScript. However, it is possible in Java to dynamically load new classes into a program, which may invalidate the result of certain analyses performed before the new code was

loaded. The traditional way to proceed in this situation would be to make no assumption in any cases that could possibly be affected by dynamic loading, and thus to limit the range of applicable optimizations. In order to avoid such a compromise, a different approach was taken in the implementation of the HotSpot Java VM. This VM is able to make inlining decisions based on currently available information, and to undo these inlining choices if dynamically loaded code contradicts them later [37]. A technique for implementing this kind of invalidation efficiently is discussed in a paper by Arnold & Ryder [38].

More recently, Pechtchanski & Sarkar have published an article about a more general approach to optimiztic optimization which integrates it into an interprocedural analysis framework for Java [39]. This system allows implementing interprocedural analyses in an optimistic way, and applying them in a systematic manner. The framework supports invalidation of analysis results when dynamic code is loaded. It also provides a mechanism to keep track of dependencies between optimizations and analysis results. The system was used to implement an interprocedural type analysis which can significantly reduce the quantity of polymorphic calls. The authors claim that their framework be could used to implement a broad range of optimistic analyses and optimizations into a unified system.

2.8 Garbage Collection

Dynamic languages rely on a technique known as garbage collection for memory management. Garbage collection means that blocks of allocated memory are automatically reclaimed by an algorithm known as a Garbage Collector (GC) once they are found to be unreachable, and thus provably no longer needed by the running program. This contrasts with languages like C and C++, which expect the programmer to manually deallocate or free blocks of memory which will no longer be used.

Garbage collection was originally invented by John McCarthy to solve some problems in LISP [40], one of the first dynamic languages. LISP, like most dynamic languages, typically allocates many objects. Basic operations such as addition of numbers or string concatenation can allocate objects to store the result. If the programmer was required to keep track of all these allocations and manually deallocate the resulting objects, the language would become extremely verbose, tedious to use, and likely more prone to memory leaks than C, as there would be many more allocations to keep track of. Dynamic languages require a GC to be usable, and because they perform so many allocations, this GC needs to perform competitively.

The simplest form of GC, which was previously used by PHP, is based on reference counting. In this system, each allocated object has an associated reference count (number of references pointing to the object) that is maintained during its lifetime. If this count ever reaches 0, the object is known to be unreachable, and can be reclaimed. This form of GC is easy to implement and predictable in its deallocation behavior. Reference counting, however, suffers

from an important flaw: it cannot detect when cyclic data structures become unreachable, and thus cannot collect them. For this reason, it needs to be supplemented with a separate algorithm that detects and collects unreachable cycles [41], as was done in more recent versions of PHP. Outside of the need for a cycle collector, the main disadvantage of reference counting is the high overhead involved in constantly maintaining the reference counts of all objects up to date.

The most well-known variants of GC are those based on mark-and-sweep (MS) and stop-and-copy (SC) algorithms [42]. MS GCs traverse the graph of references in a program starting from a set of known roots (e.g.: global variables and stack frames) and mark all reachable objects. The unreachable objects are then deallocated. SC type algorithms operate based on an approach where the reachable objects in the heap are copied to a new heap, and only the live, reachable objects are traversed by the garbage collector. SC has a clear advantage over MS in that MS needs to scan the entire heap, whereas SC only traverses the live objects. SC also has the advantage that it can be coupled with an allocator that allocates object by simply incrementing a pointer to the next available memory slot. This makes for fast allocations and can help increase the locality of allocated data structures. However, in their simplest forms, both MS and SC suffer from a potential weakness in that collections can take a noticeable amount of time, which results in the program being visibly stopped while the GC runs. This can be problematic for interactive applications such as games, simulations and music playback.

To combat these "embarassing pauses", incremental GCs have been devised. These attempt to interleave small amounts of collection work, generally triggered at allocation time, with normal program execution [43]. The main downside of this approach is that the GC algorithm needs to take special care to ensure that it cannot interfere with the executing program and vice-versa. Another possible approach is that of generational GCs [44]. This technique divides allocated objects into multiple generations based on their age, that is, how many collection cycles they have survived thus far. Generational GCs are based on the following heuristics:

1. Most recently allocated objects will be short-lived.
2. Long-lived objects are likely to remain live for the foreseeable future.
3. References from older objects to newer objects are infrequent.

In a generational GC, the youngest generation is known as the "nursery". This generation is usually small in terms of memory space, and is collected the most frequently. Older generations are collected more infrequently, when they exceed their memory quota. This permits the GC to execute rapidly for most collection cycles. An added complexity of this type of GC is that lists of references from older generations to younger generations must be maintained. This can be made faster by relying on hardware page protection to notify the GC when such references are created.

Because dynamic languages make many allocations, special care needs to be taken to optimize them. Some possible optimizations are as follows:

1. Eliminate allocations that can be shown unnecessary. For example, if the language typically allocates floating-point numbers on the heap, it may be possible to avoid this for intermediate floating-point results.

2. Allocate small objects on the stack instead of the heap when their lifetime is bounded by the activation of the current function.

3. Group multiple allocations within a single function together, so as to minimize allocation overhead.

4. In a generational GC, detect sites where long-lived objects are allocated and allocate these in older generations directly.

Today, with the arrival of multi-core processors and the increasing importance of parallelism, it is becoming less and less acceptable to have single-threaded GCs which completely stop program execution to perform a collection. The trend in research seems to be going towards GCs that can benefit from parallelism themselves, either by parallelizing the GC process [45], by having an incremental GC run in a separate threads while the program continues executing, or both. At this time, it seems that marrying all the desirable GC characteristics together (high throughput, short pause times and high-scalability) is a rather difficult problem deserving of more research.

2.9 Metacircular Systems

Tachyon, the compiler I will be using in my research, is said to be metacircular. That is, Tachyon compiles JavaScript code, but is also written in JavaScript, and so it is able to compile itself. Metacircularity is fairly easy to achieve in a static compiler, as this one only needs to generate code for the target architecture. It is somewhat more complicated when one considers dynamic languages, as these require special runtime support. Furthermore, dynamic languages usually do not expose low-level memory access and other features required to perform JIT compilation.

A well-known metacircular JIT compiler is the Jalapeño [46] Java VM from IBM (now known as JikesRVM). The design of Tachyon was partly based on that of this VM. Their effort has exposed some potential solutions to the problems involved in bootstrapping such a compiler and getting it to run independently from its host platform. In the case of Jalapeño, the Java language needed to be extended to provide lower-level control over the target architecture, and despite Java's lazy loading of classes, a set of core classes comprising the basis of the compiler needed to be preloaded in a binary image for the system to run. A similar metacircular system called Maxine VM is currently being developed at Sun Microsystems [47].

There are few efforts at metacircular implementations of dynamic language VMs. This may be because metacircular VMs are rare to begin with. It may also be due to the fact that dynamic languages are thought to result in much slower code, and VMs need to perform well in terms of compilation time. Some notable implementations are Klein, which is a SELF VM written in SELF [48], and PyPy, which could be described as a VM generator for Python written in Python [49].

Some of the objectives behind the Klein project were to favor maximal code reuse and allow exploratory, incremental programming. Some of its more interesting features were a reflective object model and support for remote debugging. The authors claim that the design of the resulting VM is simpler and that their approach can yield faster development than would be possible with a static programming language [48]. They also say that these benefits came at the expense of performance, however. The Klein VM was unfortunately never fully completed, but is now open source.

PyPy uses a rather different approach from other metacircular VM projects. It does not directly compile python code into executable code. Rather, their approach involves describing the semantics of a bytecode interpreter for a programming language (Python, for instance), and generating a VM (e.g.: generating C code) that supports this language from the description. They are able to improve on the performance of the raw bytecode interpreter by applying a tracing JIT compiler to it [50]. Their system now supports most of Python and significantly improves upon the performance of the stock CPython distribution.

Chapter 3

Objectives

3.1 Optimistic Optimization

Optimistic optimization is a fairly new optimization technique that remains relatively unexplored. I believe it has the potential to complement existing techniques and bring dynamic languages closer to the level of performance of static languages than has been possible until now. As part my thesis, I intend to implement a framework to test optimistic optimizations in the Tachyon JavaScript VM. I will attempt to demonstrate that optimistic optimizations can be applied in a systematic manner, that is, optimistic assumptions can be used as the basis for optimizations in the VM as part of a unified framework, resulting in significant performance gains due to efficient specialization of code.

The following subsections contain an explanation the nature of optimistic optimizations and their potential benefits, as well as an example of the kinds of optimizations that can be achieved. An analysis of the performance implications of optimistic optimizations is also provided.

3.1.1 Powerful Optimizations

As mentioned in section 2.7, some forms of optimistic optimization have already been applied to dynamic languages. Notably, trace compilers are having great success right now in their application to languages such as Python, JavasScript and Lua. It seems that these have become the dominant approach in the dynamic language optimization realm. However, we consider that these only exploit a small part of the possibilities offered by optimistic optimizations.

An important limitation of trace compilers is that they cannot eliminate a large part of implicit conditional tests. Such compilers generally use little or no static analysis of code and have not yet, to our knowledge, been combined with interprocedural analyses. They are purely based on the idea of optimizing the most frequently used linear paths through the loops of a program, and only perform local analyses, if any analyses are used.

This brings a crucial limitation, that is, trace compilers, by themselves, cannot determine with certainty the type of a global variable, or inline a function without first verifying the identity of the function to call. It also seems that, at this point, trace compilers do not optimize code outside of loops. This may be problematic, because it seems that most JavaScript programs are heavily event-driven, and long-running loops are uncommon [27].

The optimization techniques I wish to explore for this thesis do not aim to replace trace compilers; rather, they are orthogonal to these. What I aim to accomplish, in fact, is to explore a different direction in terms of optimistic optimization. I will to apply these in a systematic manner, while integrating profiling and analysis techniques in our system, similarly to the work of Pechtchanski & Sarkar [39].

The semantic of dynamic languages generally implies the presence of implicit dynamic tests (type tests, for example) in the generated code. Unfortunately, these tests are inserted in many locations, and can have a high cost in terms of execution time. The current dynamic language optimization techniques are simply not able to eliminate many of these tests, and instead generate code that is optimized for the most frequent execution path.

I propose an approach which implies generating code in an optimistic manner, which includes little or no tests of this kind. The tests are instead replaced by conditional guards serving to verify that the invariants on which optimistic optimizations are based are maintained. If these invariants are invalidated, the optimistic code is recompiled without the broken optimizations. The main advantage of this approach is the potential to eliminate most conditional tests, replacing them by guards which are not on the critical path, and thus do not have nearly as much performance impact. Another advantage is that the most frequently executed code generated by our system could be much more compact and thus reduce the number of instruction cache misses.

3.1.2 Code Example

Figure 3.1 shows an example of a simple JavaScript program. The `sum` function is called to compute the sum of the numbers in a list. The global variable `zero` is used to initialize the sum. There is also a function `f` which, if called, may redefine the `zero` variable. The semantics of this program may appear obvious, but this program is actually difficult to optimize efficiently.

For one, additional code could be dynamically loaded later on. We do not know for a fact that the sum function won't be called on lists containing other types of elements than numbers. If the list were to contain strings, for example, then the sum operation would perform string concatenation. We also have that the type of the `zero` variable could change. Thus, dynamic type checks need to be inserted into this program to properly implement its semantics.

A sketch of the type checks needing to be inserted is provided in figure 3.2. Here, the types of the list element and the sum variable are tested at each loop iteration, before the addition operation is performed. If both values are numbers, then a number addition is performed. If either of the values is some-

```
var zero = 0;

function sum(list) {
   var total = zero;
   for (var i = 0; i < list.length; ++i)
   {
      var t = list[i];
      total = total + t; // Addition or concatenation
   }
   return total;
}

function f(x) { zero = x; }

print(sum([1,2,3,4,5]));
```

Figure 3.1: A simple JavaScript code sample.

```
var zero = 0;

function sum(list) {
   var total = zero;
   for (var i = 0; i < list.length; ++i) {
      var t = list[i];
      if (typeof sum === 'number' && typeof t === 'number')
         total = numberAdd(total, t);
      else
         total = genericAdd(total, t);
   }
   return total;
}

function f(x) { zero = x; }

print(sum([1,2,3,4,5]));
```

Figure 3.2: Sketch of dynamic checks inserted in the sum program.

```
var zero = 0;

function sum(list) {
   var total = zero;
   for (var i = 0; i < list.length; ++i) {
      var t = list[i];
      total = numberAdd(total, t);
   }
   return total;
}

function f(x) {
   zero = x;
   if ((zero instanceof Number) === false)
      sum = eval(sum);
}

print(sum([1,2,3,4,5]));
```

Figure 3.3: Sketch of optimistic guards inserted in the optimized sum program.

thing other than a number, then a slower, general-purpose addition operation is performed. This may turn out to be a string concatenation, for example.

Because additional code can be dynamically loaded at any point, even a sophisticated static analysis cannot get rid of all the dynamic type checks in such a program. Performing a dynamic type check at each loop iteration is inefficient, especially if all the `sum` function is ever used for is the addition of number values. It may be obvious to the programmer who wrote this code that these type checks are useless, but such a fact is not obvious for a traditional JavaScript compiler to prove conservatively.

A VM with a JIT compiler, however, can observe the types of all variables and function arguments in a program while it is executing. A VM using optimistic optimizations could inspect the state of the program after it has just been initialized, and assume that the `zero` variable will not be redefined to be anything other than a number and that `sum` will only ever be called on lists of numbers. In this case, an optimized version of the program could be compiled, as illustrated in figure 3.3.

This optimized program contains no dynamic checks inside the loop and simply performs number addition at each iteration. Instead of dynamic checks, a guard has been inserted in the `f` function, where the `zero` variable is likely to be redefined. If the assumption that `zero` is always a number turns out to be violated, the `sum` function should be recompiled to remove the optimization based on this assumption. Similarly, the function should also be recompiled if the `sum` function is called with an argument other than a list of numbers.

The idea here is that we can use information we observe about the current state of a running program to optimize it, optimistically assuming that some

properties will likely hold. Costly dynamic checks can then be replaced by guards that are executed less often, and are thus incur less of a run time penalty. In theory, it may be possible to completely eliminate such guards if we could infer that an assignment to `zero`, for example, could only assign a number into this variable. The guards only need to be inserted in areas of the program where we suspect some optimization assumptions may be broken.

3.1.3 Compilation Model

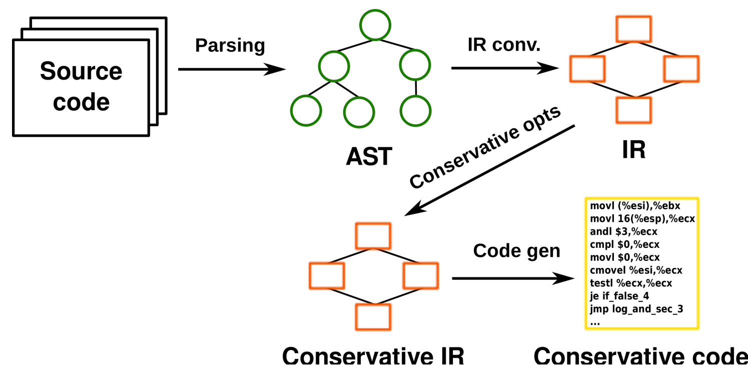

Figure 3.4: A pipelined compilation model.

In a simplistic method-based JIT compiler, the compilation phases traditionally follow a pipelined model, as illustrated in fig. 3.4. In this model, source code is parsed into some intermediate-representation. This IR is then analyzed and optimized essentially as would be done in a static compiler. This means that the optimizations performed must be guaranteed to be valid under all circumstances. The optimized IR is finally compiled into machine code. Some possible improvements upon this model include lazy compilation, the addition of different optimization levels to be used for hot and cold code, and possibly making use of profiling data to favor hot execution paths within methods.

There are many possible ways to integrate optimistic optimizations in a compiler. My research goal is to integrate them in such a way that optimistic optimizations are systematically applied. This requires the source code to be analyzed and instrumented so as to find useful properties that can be optimistically assumed to hold and used for optimization. In my case, I am mostly interested in type information, as I believe this is one of the areas where dynamic language optimization usually lacks, and type information forms the basis of the majority of optimizations.

Optimistic optimization suggests a different compilation model that integrates analysis and invalidation. One possible such model is illustrated in in fig. 3.5. In this compilation model, source code is compiled and translated into IR. Traditional conservative optimizations are also applied to this IR, as be-

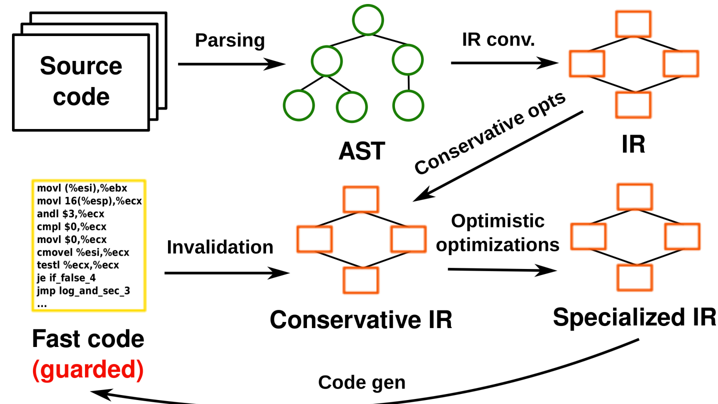

Figure 3.5: An optimistic compilation model.

fore. The difference, however, is that the IR is then analyzed and optimized in an optimistic way. Optimizations are performed that assume certain program properties found through analysis will remain valid. Guards are also inserted into the IR to detect violations of these assumptions.

Machine code can then be generated from this specialized IR, which should hopefully be much more aggressively optimized than would be possible with a traditional JIT compiler. The optimized machine code includes the safety guards required by its optimizations. As such, if optimistic assumptions are violated, this can be detected, and the machine code can be deoptimized (optimizations can be disabled). In our hypothetical compilation model, this results in the compiler reverting sections of the code (i.e.: whole methods) back to a conservative IR, so that they can be analyzed again, in light of the violated assumptions, and then reoptimized.

3.1.4 Performance Analysis

In a simple JIT compiler that compiles functions only once without optimizations, the execution time of a program could be modeled as in equation 3.1, where C_{noopt} represents the time spent compiling the program without optimizations, and E_{noopt} represents the execution time of the compiled program. Note that compilation and execution may be interleaved if the compiler uses lazy compilation, but this does not change this equation.

$$T_{noopt} = C_{noopt} + E_{noopt} \tag{3.1}$$

If we assume that we are instead using a compiler which performs optimistic optimizations, we must add new terms to our equation, as in equation 3.2. We can imagine that the compiler will spend some time analyzing code (A term). It will then compile an optimized and guarded version of this code based on

this analysis (C_{opt}) and begin executing it (E_{opt}). The guards added to the optimized code will incur some execution time penalty G. If some of these guards are tripped, the optimized code may be recompiled (R term).

$$T_{opt} = A + C_{opt} + E_{opt} + G + R \tag{3.2}$$

Since we are trying to optimize for performance, we would like to have $T_{opt} < T_{noopt}$ (see equation 3.3). The other terms of the equation, however, are difficult to quantify. Notably, E_{opt} represents the total execution time of the optimized program. Since the program may be recompiled during its execution, however, the set of optimizations and guards the program is using may change over time.

$$A + C_{opt} + E_{opt} + G + R < C_{noopt} + E_{noopt} \tag{3.3}$$

If we assume that, for the sake of this analysis, as the program gets recompiled, optimizations only get disabled, and that some minimum level of optimization is eventually reached, we have that, the longer the program executes, the closer the execution time of the program becomes to the execution time the program would have had, had it been executing using the minimum set of optimizations from the beginning (see equation 3.4). Hence, in the limit, we want the property shown in equation 3.5.

$$\lim_{t \to \infty} E_{opt} = E_{minopt} \tag{3.4}$$

$$A + C_{opt} + E_{minopt} + G + R < C_{noopt} + E_{noopt} \tag{3.5}$$

Equation 3.5 is still somewhat difficult to analyze because we do not know how to quantify the A and R terms. The assumption we are making, which we will call the recompilation hypothesis, is that the number of recompilations a program will experience will eventually reach an upper bound. This is because we start from some highly optimistic assumptions about the program, and pessimize them as the program executes and some of these are found to be broken. Since there is only a finite set of possible execution paths through a program's code, and a finite set of assumptions to be invalidated, the program cannot be recompiled an infinite number of times. Thus, we expect that the number of recompilations over the execution of a typical program will follow a curve like that illustrated in figure 3.6.

Note that programs may have multiple execution phases, and may not execute parts of their code until long after their execution began. As such, the number of recompilations may follow more of a stepwise curve. Nevertheless, we still expect that this will reach a plateau as there are less and less unexplored regions of the code to be executed.

Because the number of recompilations should eventually reach an upper bound, we expect that similarly, the analysis and recompilation time will also reach an upper bound. Thus, in the limit, several terms on both sides of equation 3.5 should converge towards constant values, as shown in equations 3.6 and 3.7.

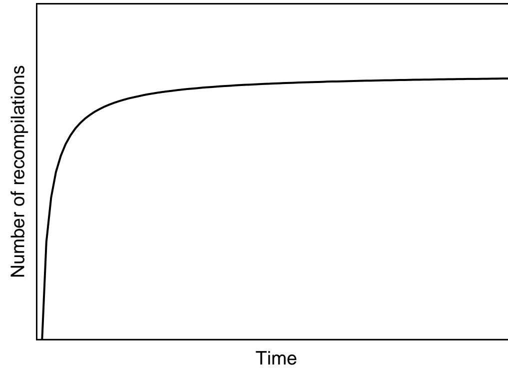

Figure 3.6: Number of function recompilations over time

$$\lim_{t\to\infty} (A + C_{opt} + R) = K_1 \tag{3.6}$$

$$\lim_{t\to\infty} C_{noopt} = K_2 \tag{3.7}$$

Because those terms are constant, they do not affect the limit of the total execution time on both sides of equation 3.5. Thus, for the execution time of the optimized program to improve over the unoptimized program, in the limit, we get the property shown in equation 3.8.

$$E_{minopt} + G < E_{noopt} \tag{3.8}$$

That is, we want the sum of the optimized execution time and the overhead time added by the guards to be less than the unoptimized execution time. This realization may seem trivial, but it indicates that it is important to minimize the number of guards and carefully place them so as to reduce their execution overhead as much as possible. Otherwise, the cost of the guards could actually make the program slower.

This performance analysis examines the limit case, however, we must also note that the analysis and recompilation times, even if they do converge to some upper bound, also need to be minimized as well. Otherwise, the program will need to execute for a long time before the cost of analysis and recompilation can be amortized.

3.1.5 Potential Limitations

The optimistic optimization model described so far may seem limited in the way it treats invalidation of optimistic assumptions. There are many ways to handle the invalidation of optimizations. The simplest is probably to simply permanently revert code containing invalidated optimizations back to its unoptimized state. This may pose a problem in some cases because performance-critical sections of programs could end up in a permanently deoptimized state.

One can imagine an image editing program which can be used to apply filters to both black and white and color images, for example. If the user first uses the program on color images, where filters have to deal with objects representing color triples, the filtering code could be optimized specifically for this case. If the user then opens a black and white image, and the filtering code becomes permanently deoptimized because the color values have a different format, this may result in poor performance until the program is restarted, even if the user goes back to editing color images.

A better strategy, which I aim to test, is to gather additional type information at the time when optimistic assumptions are invalidated, and use this information to compile a new version of the optimized code which can deal with the new state of the executing program. In the case of our image editing program, for example, once it is found that a filtering function is to be applied to a new kind of color value, it may be advantageous to compile a new version of the filtering function (i.e.: using procedure cloning) optimized specifically for the new kind of image, and ensure that applications of the function to this new kind of image go through the new version.

The programs that will benefit the least from optimistic optimizations are those which make maximal use of the dynamic features of the language being optimized. Imagine a program with a loop operating on a list of elements of many possible types, such as string, booleans, integers, floating-point numbers and objects, for example. In this case, the only way to optimize the loop is to expand its body to dispatch to every possible type that can be received. Essentially, the optimized program will resemble the naively interpreted case, because there is no other way to deal with this problem. Still, if this loop does not treat every possible type, the optimistically optimized version may still offer better performance. Fortunately, we believe that such gratiutous uses of dynamic language features are rare in practice.

3.2 Success Criteria

Compiler research is experimental in nature. It is largely a matter of devising strategies based on published results, prior experience, and intuition. Hence, what constitutes a positive result may be ambiguous in some cases. Nevertheless, as a computer scientist, I aim to approach this issue in a scientific manner. In this case, this means making predictions about the properties of optimistic optimizations as I will be implementing them, and establishing measurable criteria for success.

Some predictions about the behavior of optimistic optimizations are given in section 3.1.4. Expanding on these, I predict that:

1. For the vast majority of programs, the number of recompilations should converge asymptotically towards an upper bound as execution progresses.

2. The performance advantage of optimistic optimizations should grow as the execution time of a program increases.

3. No real-world programs will invalidate all of their optimistic optimizations.
4. Because of the previous predictions, the performance cost associated with analysis and recompilation will be amortized if programs execute for a sufficiently long time.

The question of how to measure the success is largely one of how well the programs optimized using our system perform. I am mostly concerned with performance in terms of execution time of the optimized programs, but other aspects, such as compilation time, are also important. If our system adds too much compilation time overhead, then it will no longer be be directly appicable to short-running programs.

Ideally, I would like to establish the following measures for success, which I believe are within reach:

1. The optimizations should cause no performance degradation (in terms of total running time) on any benchmark program with an unoptimized running time of 2 minutes or more.
2. The optimizations should yield an average performance improvement of at least 50% accross our entire benchmark suite, as compared to our own system with optimistic optimizations disabled.
3. The average running time obtained over the benchmark suite should be hopefully better than, but at worst 50% slower than that of commercial JavaScript VMs such as Google Chrome and SpiderMonkey.
4. In the end, our system should be optimized so as to avoid making too many optimistic optimizations that become invalidated. On average, no more than 30% of such assumptions should be invalidated across the benchmark suite.

My hope is that using optimistic optimizations will make it possible to generate code that performs even better than commercial JavaScript VMs. However, considering that these commercial VMs are maintained by large teams of dozens of programmers who have already had years to implement many optimizations and that I have limited time to complete my Ph.D., it may be difficult to compete with the performance levels of these highly-tuned implementations. As such, I have set the goal of obtaining performance levels at worst 50% slower than theirs, which I believe to be within my reach. I do, however, plan to do everything within my power to beat their performance levels.

The question then remains as to which benchmarks to use to test optimistic optimizations. Many JavaScript benchmarks commonly used today are either very small (sometimes as short as 3 lines of code) or very short running (sometimes less than 10ms). I do not believe such benchmarks are representative of real-world programs. As such, I would like to limit the selection to benchmarks Tachyon can support, and which are complex enough and run long enough to give us optimization opportunities.

Tachyon currently supports enough of the string and array libraries of JavaScript to support common uses of the said libraries. Support for the Math library and floating-point numbers should be completed within the next few months. However, support for regular expressions is currently rather basic. As such, I will probably not include benchmarks that principally aim to test the performance of the regular expression subsystem. I cannot guarantee that I will have time to integrate Tachyon into a web browser. Hence, benchmarks which rely on the HTML Document Object Model (DOM) cannot be included.

It is difficult to establish what makes a benchmark "representative" of JavaScript in general. Some attempt can be made at extracting code from real web applications. However, the landscape of web applications is already very diverse and includes code written in many different styles. In order to try and make our benchmark suite as unbiased as possible, I propose to follow the these rules in assembling it:

- The benchmark suite should include at least 20 benchmark programs.
- Benchmarks may not be excluded solely because optimistic optimizations are not expected to perform well on them.
- Benchmarks that are stress tests of JavaScript libraries (eg: array, string or regexp stress tests) should be avoided as they do not resemble real code.
- An effort should be made to include (open source) code from real web applications into our benchmark suite.

Provided that the previous guidelines are followed in assembling our benchmark suite, and that our success criteria are met, the optimistic optimizations framework will be considered to be successful at optimizing JavaScript code. If the success criteria are not met, I will adjust my design to improve its performance.

3.3 Metacircular Implementation

The Tachyon VM has a metacircular implementation (see section 2.9). This could prove to have interesting benefits in that it may simplify the design of the compiler itself and make it easier to maintain. However, it should also offer interesting performance benefits by exposing new optimization opportunities that are harder to exploit in non-metacircular systems.

One such advantage is that most of the Tachyon runtime system is written in a JavaScript dialect that is compiled by Tachyon itself. Contrarily to other systems, there are very few parts of the system that are written in C. This is advantageous because it means the primitives used by programs compiled and run in Tachyon are not opaque to Tachyon. Our compiler can easily analyze, inline and optimize the said primitives in the context of the calling code. This is more difficult in the context of a system which makes calls to many primitives

written in another language, because foreign functions largely have to be treated as black boxes, which become optimization boundaries.

Another important benefit is that Tachyon may eventually be able to optimize itself as it runs. By gathering profiling data about its own internal workings, Tachyon should be able to adapt to the program it is currently running, or different classes of programs. This should make the recompilation and analysis process faster. It may be difficult to conceive that a JIT compiler could dynamically recompile and replace parts of its own code as it is running, but I believe this should be technically feasible, so long as the code replacement mechanism cannot interfere with itself (e.g.: by trying to replace itself, or by triggering an infinite loop of recompilations).

3.4 Long Term Perspectives

It has already been demonstrated that JIT compilers can use dynamic recompilation and optimistic optimizations to improve the performance of code [37, 38, 39]. However, optimistic optimizations are speculative, and may not always improve performance. At this point, many questions remain as for how to choose which optimizations should be optimistically applied, what conditions should be used to prevent too many recompilation from happening, and whether or not problems such as long pause times caused by many simultaneous invalidations can be easily avoided.

Since it is possible that Tachyon may take some time to analyze and optimize a program to its peak potential, an interesting perspective to consider is that it may be useful to combine optimistic optimizations with a code serialization system, so that optimized versions of programs can be saved for later use. In the context of JavaScript, it is often the case that programs are run for very short periods of time, which may not make it worthwhile to spend time analyzing optimizing the said programs. However, if these programs could be progressively optimized over several runs and saved for later use, it would make optimistic optimizations applicable to a wider range of uses. This ties in with the work on continuous optimization by Kistler & Franz [24].

Chapter 4

Conclusion

4.1 Current State of the Project

At this stage, a significant portion of the infrastructure work needed for this thesis has already been completed. Tachyon is already a real, working VM which correctly supports most of the ECMAScript 5 JavaScript specification. The proper workings of the compilers have been validated using a custom bank of many unit tests, and by having Tachyon compile and run itself. In addition to this, Tachyon provides an an interactive user environment (shell).

My work thus far has focused on implementing the front-end part of Tachyon. For this, I have designed an SSA-based intermediate representation which I believe will lend itself well to analyses and transformations. The front-end already includes many commonplace optimizations, such as Sparse Conditional Constant Propagation (SCCP), Common Subexpression Elimination (CSE) through value numbering, as well as strength reduction and peephole optimizations.

The front-end also includes some JavaScript extensions which make it possible to express low-level constructs needed to implement a compiler, such as raw memory accesses, calls to primitive C functions and the specification of data object memory layouts (i.e.: how JavaScript objects are stored in memory). These extensions are currently used to implement the primitives of the Tachyon runtime.

Since this project has begun, it has also grown in size significantly. Starting with just Marc Feeley, Erick Lavoie and myself, the project now involves an additional participating professor (Bruno Dufour) and 5 new student participants. Tachyon has been briefly presented at the CASCON conference, and has attracted industrial attention. Mozilla has expressed interest in Tachyon and is currently funding our research group. The Tachyon project is also now officially open source, and is publically available through github. Finally, a publication about the design and bootstrapping of Tachyon will be appearing at the Dynamic Language Symposium 2011 [51].

4.2 Thesis Time Table

The following is an outline of the work I anticipate needs to be completed in order to achieve the goal of my thesis. The steps described are based on my current understanding of the problems involved. The times given are approximate figures based on my experience working on my M.Sc. thesis and the implementation of Tachyon.

Completing the Groundwork

I am currently working on preparing for the final part of my predoctoral examination which will be completed in July or August 2011. I am also involved in the writing of a publication about the Tachyon system for the DLS 2011 conference. I plan to take the rest of the summer of 2011 to finish these tasks and to complete some infrastructure work that will facilitate my thesis work in Tachyon. I anticipate that I will begin working on tasks more directly related to my thesis in September or October 2011.

Type Lattice Conception

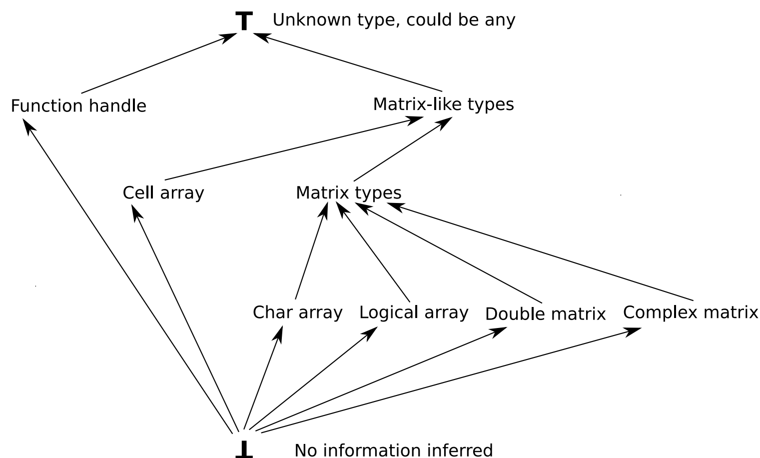

Figure 4.1: A lattice for MATLAB types.

The first step prior to beginning my work analyzing the types of values in JavaScript program should be to establish a theoretical framework to base this type analysis upon. I plan to design a type lattice to approximate those types. This lattice will likely be similar to those described in my M.Sc. work [5] and in the Jensen et al. paper [6]. See figures 4.1 and 4.2 for an illustration of the lattice I designed to approximate MATLAB types. The lattice design will reflect a compromise between the level of expressiveness needed to effectively optimize code and the computational complexity required to manipulate it.

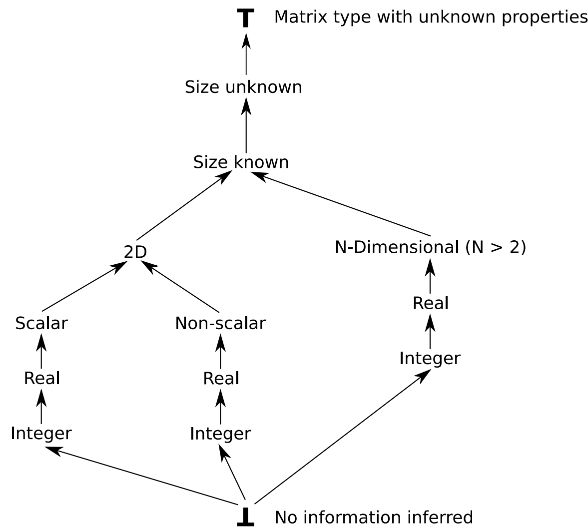

Figure 4.2: Sub-lattice for MATLAB matrix types.

I intend to design the initial lattice based on my prior experience with Tachyon and the ECMAScript 5 specification. I will attempt to anticipate what information will be useful in optimizing programs. The lattice I design can always be modified later if it does not provide adequate information. I anticipate that this first step of the work can be completed within a month. It will be interleaved with the design of the type analysis system.

Type Analysis System

I will design a type analysis system based on abstract interpretation (type propagation), as was described in my M.Sc. thesis. This will essentially be a system to locally analyze methods of a program and try to map each value to a type in the type lattice. This kind of analysis is somewhat similar to constant propagation and I believe it could potentially be combined with it, in an SCCP-like algorithm. I anticipate this step to take a few months. An early prototype can probably be completed before December 2011.

Type Specialization

Once I have a type analysis framework in place, the next logical step seems to be to try using this information to optimize JavaScript programs, including Tachyon itself. For this, I intend to design a framework to specialize the implementation of runtime primitives based on information available about local types. This information will initially be derived purely from type analysis.

The initial optimizations performed may simply consist of prioritizing certain

types and inlining parts of the implementation of primitives. These would not constitute optimistic optimizations per-se, but will give me an idea of the next steps to take in my investigations. I anticipate that a simple type specialization framework could be completed in 1 to 2 months, before March 2012.

Type Profiling

I believe that in order to get a sufficiently complete picture of the types in JavaScript programs, a combination of static analysis and profiling will be necessary. As such, I plan to design a profiling system to capture the types that cannot be inferred through analysis. These can be types of function arguments, object fields, global variables and closure variables. Profiling types will allow me to experiment with techniques such as type feedback for optimization [19, 18].

Profiling of types should not be difficult to implement, but, in order for this to work, Tachyon will need to be fully boostrapped and run independently from its current host platform (Google V8), so that it can be called by the running program as new type information is captured. It may also prove to be challenging to minimize the performance impact of profiling. I am hoping that the type analysis can help eliminate the need for most of the profiling. I anticipate that this step will take several months, possibly up to 6. I aim to complete it by the end of the summer of 2012.

Optimistic Optimization Framework

The last step of my thesis project will be to combine type analysis, type profiling and type specialization into a framework that is able to perform optimistic optimizations. This means that the system will be able to optimize code based on type analysis results with the possibility that some optimizations may be disabled later on.

This will require implementing a system to disable optimizations, either through code-patching or some form of recompilation and on-stack replacement. It will also require building a system to keep track of what optimizations depend on what optimistic assumptions and insert guards to detect when these assumptions may be violated. This may be the longest step of the project, and may require modifications to the type analysis and profiling systems. I hope to have a working version by January 2013.

Other Work

Other work will need to be completed while I advance on my thesis project. It is likely that I will participate in the writing of other publications with the Tachyon team. I will also need spend some time working on maintenance and debugging of the Tachyon compiler. This may add extra overhead to my thesis work. Since my research is somewhat exploratory in nature, it is also likely that I may need to reexamine my approach and rethink some aspects of my plans, which may result in additional delays.

4.3 Future Programming Languages

Dynamic languages have been rapidly gaining in popularity, particularly in the domain of dynamic web applications. These gains can in large part be attributed to the advantages dynamic languages offer in terms of development productivity, rapid deployment and portability. Seeing that university undergratuates are increasingly exposed to such languages, and that dynamic languages are also gaining foothold in other areas (e.g.: MATLAB being used for scientific computing), it is conceivable to imagine that perhaps, someday, they will largely replace static languages in more "traditional" application domains, such as servers, operating systems, videogames and high-performance computing.

Until recently, applying dynamic languages to these domains would have seemed quite far-fetched because of the large performance disadvantage they suffered from. However, seeing that big leaps have been made (and will likely continue being made) towards improving their performance, this becomes increasingly realistic. In a way, this change is already happening today. Many desktop applications and videogames already implement a plugin mechanism using dynamic languages. One can think of the Firefox web browser, for example, whose rendering engine is largely implemented in JavaScript.

One significant caveat to this forthcoming change, however, is that there remain many unsolved issues with dynamic languages. For one, most current such languages have little or no support for any kind of parallelism. Another important issue is that such languages, providing less support for static verification than static languages, can be more unpredictable and harder to debug. Programmers have often accused languages such as C++ of being highly unpredictable because of the support for low-level features such as pointers, but in a language such as JavaScript, many trivial bugs that would be instantly detected by a type system can go unnoticed for a long time.

Many of today's dynamic languages were simply not designed with performance in mind, but rather with the view that these languages would never be used for anything beyond almost trivial "scripts". Because of these issues and their increasing popularity, I believe dynamic languages are deserving of more research. It is likely that new dynamic languages will need to be designed to meet the safety and scalability needs of future applications. In this regard, I believe that issues such as ease of optimization, parallelism and security should be given more importance in the design of dynamic languages of the future.

Taking compiler design into account, I believe it is possible to make implementation choices that can simplify optimization while making language semantics more consistent and predictable, and thus, easier to understand. It is my hope that my research work will help better optimize the dynamic languages of today, and perhaps provide some insight into how to design better ones for tomorrow.

Bibliography

[1] J.K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[2] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object- oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, 1989.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[4] F. Logozzo and H. Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation–Application to JavaScript Optimization. In *Compiler Construction*, pages 66–83. Springer, 2010.

[5] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Compiler Construction*, pages 46–65. Springer, 2010.

[6] S. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. *Static Analysis*, pages 238–255, 2009.

[7] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[8] T. Zhao. Type inference for scripting languages with implicit extension. FOOL, 2010.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[10] M. Furr, J.D. An, J.S. Foster, and M. Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1859–1866. ACM, 2009.

[11] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. *ECOOP 2010–Object-Oriented Programming*, pages 126–150, 2010.

[12] D. Jang and K.M. Choe. Points-to analysis for JavaScript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1930–1937. ACM, 2009.

[13] D. Detlefs and O. Agesen. Inlining of virtual methods. *ECOOP99Object-Oriented Programming*, pages 668–668, 1999.

[14] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):135–150, 1993.

[15] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, pages 146–160, 1989.

[16] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 137–148. ACM, 1996.

[17] S. Soman and C. Krintz. Efficient and general on-stack replacement for aggressive program specialization. In *International Conference on Programming Languages and Compilers (PLC), Las Vegas, NV*. Citeseer, 2006.

[18] R. Muth, S. Watterson, and S. Debray. Code specialization based on value profiles. *Static Analysis*, pages 340–361, 2000.

[19] U. Hölzle and O. Agesen. Dynamic versus static optimization techniques for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):167–188, 1995.

[20] J. Whaley. *Dynamic Optimization through the use of Automatic Runtime Specialization*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 1999.

[21] M. Furr, J.D. An, and J.S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300. ACM, 2009.

[22] K. Williams, J. McCandless, and D. Gregg. Portable Just-in-Time Specialization of Dynamically Typed Scripting Languages. *Languages and Compilers for Parallel Computing*, pages 391–398, 2009.

[23] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM, 2004.

[24] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.

[25] A. Holkner and J. Harland. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 19–28. Australian Computer Society, Inc., 2009.

[26] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12. ACM, 2010.

[27] P. Ratanaworabhan, B. Livshits, and B.G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, page 3. USENIX Association, 2010.

[28] D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 97-27, University of Arizona, October 1993.

[29] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[30] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[31] A. Gal, C.W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153. ACM, 2006.

[32] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M.R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478. ACM, 2009.

[33] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2009.

[34] Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation

in a tracing jit. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 43–52, New York, NY, USA, 2011. ACM.

[35] R. Sol, C. Guillon, F.M.Q. Pereira, and M.A.S. Bigonha. Dynamic Elimination of Overflow Tests in a Trace Compiler. 2011.

[36] J. Ha, M.R. Haghighat, S. Cong, and K.S. McKinley. A concurrent trace-based just-in-time compiler for single-threaded JavaScript. *PESPMA 2009*, page 47, 2009.

[37] T. Kotzmann, C. Wimmer, H. M
”ossenb
”ock, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008.

[38] M. Arnold and B. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. *ECOOP 2002Object-Oriented Programming*, pages 31–78, 2006.

[39] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: a framework and an application. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 195–210. ACM, 2001.

[40] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[41] D. Bacon and V. Rajan. Concurrent cycle collection in reference counted systems. *ECOOP 2001Object-Oriented Programming*, pages 207–235, 2001.

[42] B. Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 87–98. ACM, 1990.

[43] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[44] A.W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.

[45] M. Wu and X.F. Li. Task-pushing: a scalable parallel gc marking algorithm without synchronization operations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE.

[46] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2010.

[47] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 59–68. ACM, 2010.

[48] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20. ACM, 2005.

[49] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[50] C.F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[51] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: an experience report. *Dynamic Language Symposium 2011*, October 2011.