DEGREE PROJECT, IN MEDIA TECHNOLOGY , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

Ahead of Time Compilation of EcmaScript Code Using Type Inference

JONAS LUND

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

Ahead of Time Compilation of EcmaScript Code Using Type Inference

Förkompilering av EcmaScript programkod baserad på typhärledning

Jonas Lund
whizzter@kth.se

DM228X Degree Project in Media Technology 30 credits
Interactive Media Technology
Degree Progr. in Media Technology 270 credits
Royal Institute of Technology year 2015
Supervisor at CSC was Vasiliki Tsaknaki
Examiner was Haibo Li

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: *www.kth.se/csc*

Ahead of Time Compilation of EcmaScript Code Using Type Inference

Abstract

To investigate the feasibility of improving performance for EcmaScript code in environments that restricts the usage of dynamic just in time compilers, an ahead of time EcmaScript to C compiler capable of compiling a substantial subset of the EcmaScript language has been constructed. The compiler recovers type information without customized type information by using the Cartesian Product Algorithm. While the compiler is not complete enough to be used in production it has shown to be capable of producing code that matches contemporary optimizing just in time compilers in terms of performance and substantially outperforms the interpreters currently used in restricted environments. In addition to constructing and benchmarking the compiler a survey was conducted to gauge if the selected subset of the language was acceptable for use by developers.

Förkompilering av EcmaScript programkod baserad på typhärledning

Sammafattning

För att undersöka möjligheterna till att förbättra prestandan vid användning av programkod skriven i EcmaScript i begränsade miljöer som förhindrar användningen av dynamiska så kallade just in time kompilatorer, har en statisk EcmaScript till C kompilator utvecklats. Denna kompilator är kapabel att kompilera program som använder en större delmängd av språket såsom beskrivet i standarden. Kompilatorn härleder typinformation ur programkoden utan specifikt inlagd typinformation med hjälp av den Kartesiska Produkt Algoritmen. Emedan kompilatorn inte är utvecklad till den grad att den är användbar i produktionsmiljö, så visar testresultat att den kompilerade koden som kompilatorn producerar matchar samtida just in time kompilatorer och har stora prestandafördelar gentemot de programtolkar som används i begränsade miljöer.
Utöver konstruktion och utvärdering av kompilatorn så gjordes en undersökning bland utvecklare för att utvärdera huruvida den utvalda delmängden av programspråket var acceptabel för utvecklare.

Contents

1 Introduction

1.1 Defining the Problem Space

JavaScript is probably the most spread computer language on the planet due to it's inclusion in web browsers and this is thus the commonly used name for it even if the core parts of it was later standardized as EcmaScript (Ecma 2011). The language started out it's life as an extension language to provide simple interactivity to web pages in Netscape browsers in the mid 90s but has kept proving itself as an increasingly capable language as developers has started using internet browsers for more and more computationally expensive tasks due to significant amount of engineering effort being put into improving performance of implementations of the language.

In the last few years there has been a push in the form of HTML5 API's to improve graphical(Canvas, WebGL, FullScreen), audio (WebAudio) and network(WebSockets,WebRTC) capabilities in browsers to enable games and other interactive applications to be distributed portably to a variety of platforms via the browser.

For game companies in-browser games for the desktop is an enormously attractive proposition as the barrier of entry for a player to start playing is very small. With the performance of JavaScript implementations always increasing and as the availability of these new API's are becoming more commonplace, we are now at a breaking point where a large amount of advanced games that were previously desktop based will be developed for the web first. For example the popular Unity game development environment is in the process of abandoning the usage of a custom plug in in favor of using HTML5 as the preferred target for web deployments.

On mobiles however this proposition is slightly different as some of these API's or the user interface is crippled due to security concerns that override the needs of game developers. In addition to this the mobile ecosystem also has a better functioning marketplace for developers wishing to monetize on games with native applications.

While there exists toolkits to port these web based games to mobiles as native applications they suffer in terms of performance as they rely on interpreters while the modern JavaScript environments uniformly rely on Just in time compilation(Hereafter JIT) processes to speed up execution. A requirement for a JIT to function is that the engine is allowed to create and change the native code during execution, however this conflicts with the premises of code signing and is thus disallowed for distributing applications for various platforms that puts user security in front of developer convenience.

Under the auspices of the authors company a lexer, parser and embryo of the used type inference system had previously been produced that showed enough promise to instigate further development of a prototype compiler capable of producing actual running code. The lexer and parser are designed after the EcmaScript standard specification. The embryonic type inference system has during this thesis work undergone major changes to make it useful in an actual compiler and much of the code has changed while the core concepts has been retained.

1.2 Purpose

This work investigate the possibilities of attaining higher performance on game code written in EcmaScript language with static ahead of time (AOT) compilation instead of interpretation in places where just in time(JIT) compilers are unavailable. Due to the way dynamic typing in EcmaScript works compared to explicitly typed languages that are usually used with static ahead of time compilation, a major focus of the work on getting the static compiler to generate high performance code lies in doing type inference to make implicit type information explicit for the backend code generator target.

1.3 Delimitations

The focus is on trying to answer questions about applicability of statically compiling game code in an EcmaScript language subset. Many constructs and behaviors of EcmaScript and JavaScript are very hard to analyze and optimize, especially in a static context and this is why the studied dialect is a limited subset of the full standardized language. In fact, some of those constructs (the *with* statement) and semantics have been or are in the process of being deprecated in current and future EcmaScript and JavaScript revisions for both performance and safety reasons so their omission should not form a major impediment for developers (Ecma-262 5th ed).

The runtime system while mentioned in places is not detailed in this text. For the tested benchmarks in this report only a very small skeleton runtime library was made that mostly consists of functions to provide the operating systems memory and timing functionality to the compiled EcmaScript code. A full runtime system would require functionality to emulate some dynamic behaviors mention but in this work the dynamic behavior was tested in other ways as detailed in the benchmarking section.

Another big factor when evaluating high level languages for game development is garbage collection. In particular garbage collection pauses can hurt immersion due unpredictable delays in execution when they are triggered. While one would need to take garbage collection into account when evaluating interactive performance this evaluation should be able to ignore the effects of garbage collection pauses when doing evaluation of raw computation speed in relation to type inference.

While there is a multitude of algorithms for type inference those will only be briefly mentioned and not exhaustively investigated in comparison to the selected algorithm as this work focuses on trying to evaluate how well code behaves when compiled with a compiler based on type inference. Without doubt a more powerful algorithm could potentially improve performance but the Cartesian Product Algorithm has already been proven to be useful and powerful in previous works and part of the hypothesis this work is based on focuses on the algorithm being suitable for soft analysis.

This evaluation also does not touch upon investigating very low level program optimization techniques such as register allocation, instruction scheduling and pipelining. As the compiler targets a C/C++ compiler or a code generation system such as LLVM to handle low level code generation, these options will be able to handle low level code generation optimizations very well while giving the compiler an appropriate abstraction focusing on high level optimizations with the usage of type inference and other program analysis.

2 Related Research

2.1 Basic Type Inference Systems

Due to performance concerns practical usage in the early ages of computing the only feasible solution was to use relatively low level languages with none or explicit typing and with the limited hardware of those times the problem of doing simple optimizations on even those languages was hard enough that most optimization efforts was focused on solving these low level problems. While languages without explicit typing existed they were usually academic interpreted experiments and thus suffered from performance problems compared to established systems of the day. (Van Emden 2014)

Around 1970 Hindley-Milner type inference was defined and later implemented in ML and provided the first steps in letting programmers use implicitly typed programs with high performance. While this approach works great for statically typed programs with declarative definitions designed for this, like with the ML language, the basic premise of constraining the types in a graph has drawbacks where dynamic types with imperative definitions that can be present for Lisp like languages as it can limit the dynamism of the language. In addition to this, analyzing types in a Lisp like language must on top of the special cases needed to handle normal call/return pairs also handle continuations, a construct powerful enough to simulate both return statements and exceptions common in imperative languages.

2.2 CPS, Expansion Theory and Soft Typing

A major enabler for further research in this field for such dynamic languages was the formulation of the Continuation Passing Style(Shivers 1988) for programs with accompanying transforms from regular syntax that is hereafter known as CPS. What this new form gave was the ability to model all control flow of a program as invocations of continuations that never need to return for all control flow rather than relying on implicit return semantics (Appel 1992). For the statement “x=y+z” a regular parsed tree would be in the form of “call set_x with the result from the computation of y+z”, a CPS transformed program on the other hand will say something roughly translated to “call compute y+z, then pass on the result of this computation as an argument to call set_x”, while this seems to imply roughly the same thing but in a longer and more convoluted way it gives compiler developers a coherent way of analyzing all program flow since they can focus on a simple chain of operations instead of a tree with contextual meaning.

Naive type analysis with only unexpanded basic types and singular code paths over nodes however proved to provide limited accuracy and thus also limited performance for fully

inferred code in Lisp systems since the functional style with very primitive data leads to user composition of data containers.

As a remedy we saw the realization and formalization of N-expansion for data and code definitions when trying to achieve exact inference of polymorphic code. This provided the definition of the k-CFA family, with non-expanded object definitions as 0-CFA (no expansion), callsite expansion as 1-CFA (identify objects by calling site), 2-CFA (identify object by callsite with the allocation environment) and higher levels defined as allowing expansion to be defined in terms of identifying and expanding something with environment of an environment formalizing k-CFA. (Shivers 1991) (Might 2014).

The problem with expansion however is that for a compiler to exactly decide types for any program is impossible without actually running the program itself, thus it would needs to handle the halting problem that is proven to be undecidable and thus a compiler will therefore require various heuristics to finish analyzing in bounded time (Shivers 1991). Even without taking the halting problem into account, full expansion of the program at every expansion level is highly inefficient in terms of memory and processing power for most programs as it is likely that it's isolated code paths that expand while most code benefits very little from it.

Notable was also that soft typing was defined by Cartwright (Cartwright 1991) as utilizing type inference improvements to speed up specific case code in dynamic systems without trying to achieve perfect inference.

2.3 Modern Precision Improving Methods

Plevyak(1988) published a work proposing an iterative method that started with 0-CFA and then expanded only problematic cases before rebuilding the flow information to keep down expansion to only the bad cases, this approach showed some improvements but keeping track of the problematic cases can present some problems for a compiler writer. (Agesen 1995)

The CPA algorithm (Agesen 1995) was introduced in conjunction to development of the seminal Self system, this algorithm sidestepped the expansion problem for code by using a slightly different approach that mostly solved N-expansion problems for function invocations by using a growing argument set on invocation nodes that matches argument sets to previous occurrences if possible instead of expanding. Since this method doesn't blindly trace in data but instead relies on what actually flows into the node for deciding expansion it is easier to implement in a stable fashion, however as it only works for managing code expansions and not data it doesn't solve all expansion problems.

More recently the CFA2 algorithm was proposed (Vardoulakis 2011), it partly tries to unify the work of Agesen and Plevyak but also relies on novel pushdown heuristics to codify and terminate expansions (in a sense Agesens CPA can be seen as a heuristic to terminate

expansion). The CFA2 work has in turn spurred new research into using pushdown automata but also other approaches such as in-analysis garbage collection to limit expansion while retaining correct analysis semantics (Might 2006).

Object sensitivity while used practically before that point in time was recently codified as an excellent indicator of object identity selection to guide type inference systems as a dual to closure environments (Smaragdakis 2011)

2.4 Improvements in Interpretation and JIT Techniques

The group working on the Self system had worked on several tracks to improve performance of their system and they did a comparison of type feedback (as through a JIT) versus type inference (with static compilation) that showed a slight benefit in terms of performance for JIT compilation but also many practical advantages for development (Agesen&Hölzle 1995). One could argue that it's partly due to this paper (But also very much due to the very dynamic nature of JavaScript on the web) that we have seen a great deal of work on improving JIT compilation compared to the work done on static compilation for dynamically typed systems ever since.

Apart from type dispatch these JIT compilers has bought in methods from the old Self system (Google 2008) and added other interesting data representation tricks (Wingolog 2011) to improve performance. But work has also started going towards more explicit type verification systems to cram out even more performance as exemplified by Asm.JS (Herman 2013) or even more expensive type inference to do this implicitly (Egorov 2013), and with the requirements due to the ever higher levels of analysis in JIT systems we are again starting to see a convergence between the JIT and static analysis fields as more and more expensive analysis are being permitted to gain performance in JIT systems.

2.5 Other Static Compiler Implementations

While almost all work on JavaScript performance has focused on JIT compilation we have seen some work on static compilation of other dynamic languages used in more conventional settings.

Cython(Cython, 2014) and Nuitka(Hayen 2014) for the Python language and similar compilers are fairly plain compilers that differ relatively little from normal compilers for explicitly typed languages by mostly transforming syntax trees into corresponding abstract operations as defined by the language. However as python is a dynamic language with much expense being spent on doing the right thing with the right type the main advantage in terms of performance is more or less just in getting rid of the dispatch loop of an interpreter and some Python specific optimizations but retains much of the performance penalty of dynamic type dispatch. Cython tries to augment the performance with specific manual type annotations while Nuitka is designed to evolve outwards by optimizing increasingly bigger

snippets of code. While the improvements in performance for generic code thanks to these optimizations are small they provide very high compatibility and an opportunity to fine tune specific code paths where needed, the downside of this approach is that it will increase the amount of code duplication and maintenance if the code is to target other targets apart from the specific compiler used.

Implementations with higher performance targets as practical applications of type inference theory was done with Starkiller (Salib 2004) and Shedskin(dufour 2006) for Python and Ecstatic(Madsen 2007) for Ruby, these all utilized Agesens CPA for type inference showing the strength of this algorithm and reporting promising performance results. Shedskin also used the work of Plevyak for data expansion as suggested in Agesens original paper. Of these systems only Shedskin has been made publicly available.

3 Method

3.1 Compiler Construction

Without any actualization of the algorithms and ideas under review, any and all research into the area would just be based on speculation. Thus an optimizing prototype compiler has been constructed based on some hypothesis about game code behavior to test the possibility of generating fast code within these assumptions. First a set of hypotheses about the code to be compiled was defined, based on that a design for the compiler was ironed out and finally implemented.

3.2 Compiler Performance Evaluation

The main goal of this evaluation was to answer questions in regards to how well a well defined subset of the EcmaScript language aimed at game development can be optimized with an ahead of time compiler compared to contemporary interpreters and JIT compilers. To test this a set of increasingly more complex pieces of benchmarks are implemented and tested within a controlled test harness to give answers about performance characteristics of the code generated by the compiler compared to established solutions using other techniques. There is also a brief review and comparison on how different C compilers handles the output of the constructed compiler since all low level optimizations are deferred to the C compilers and thus a brief review will need to be done to see how this impacts the results compared to the established solutions.

3.3 Language Subset Evaluation

Since the compiler has a restricted set of capabilities amounting to a subset of the language to attain performance, an important question to answer was how well such a subset conforms to the set assumptions about real world usage of the language. Also the defined subset might present developers with benefits in terms of better error checking than the standardized language.

4 Compiler Design

This chapter describes the design of the implemented compiler, first by presenting the main motivations and direction in chapter 4.1, then by a brief conceptual overview of the internal compiler workings in chapter 4.2 and finally by presenting details of the main components in the rest of chapter 4.

4.1 System Design Considerations

The main goal in the design of the system is to provide a practical compiler for porting games and similar applications written for the web platform to specifically be compiled to native applications. As such the priorities are firstly general compatibility and secondly performance. Even if compatibility is the primary goal it is impossible to attain perfect compatibility with an ahead of time compiler due to constructs such as *eval*, thus such border cases that are impossible to analyze and optimize are left out.

The focus for most works listed in related research chapter is to attain perfect type information to generate optimal code execution speed for all possible cases by resolving optimal types in every possible case. However for this compiler it isn't necessary to attain perfect information due to three reasons presented below.

The first reason is because the primary priority of aiming this compiler as a tool for developers creating regular application code rather than those developers that are writing performance oriented libraries and container constructs exclusively for high performance computing. So while games often require high performance, it is acceptable to fall slightly short in terms of performance for complicated cases as long as the performance is close enough in those parts of the code that takes up a majority of the running time.

The second reason is that while most game developers require their compiler to produce well performing code they are usually not averse to doing manual performance tuning when needed to achieve performance goals. Thanks to this, most of the problems of slow and expensive border cases isn’t a critical problem as long as the tooling and documentation enables developers to avoid pitfalls or track down problems as they arise.

The third reason is that the target language is EcmaScript. This language, just like the implementations of Python and Ruby that used CPA before relies on the fact that objects are part of the language itself, thus creating a suitable level of abstraction for the compiler analysis. Since objects in these languages usually contain more information in terms of various properties, it can be easier to analyze their contents compared to Lisp and similar languages. In Lisp and similar languages, objects and containers are usually implemented using lower level constructs of the language itself such as list cells, making it necessary to

analyze these lower level constructs and providing an abstract view of objects. Languages with first class objects on the other hand gives us the opportunity to treat objects as whole entities in terms of analysis and thus reducing the need for higher data accuracy since the programmer has already provided an implicit link between points that needs analysis in the forms of the objects themselves. In addition to first class objects the fact that EcmaScript Arrays functions as variable size vectors in other languages and that regular Object literals are commonly used as hash tables gives the analysis system further hints about object relationships

To expand on this third reason we will look at a couple of hypotheses about how game code in EcmaScript usually behaves. These hypotheses are based on the authors experiences in writing EcmaScript code together with results reported in the research literature mentioned earlier.

The hypotheses are:

- Runtime code loading is limited or can be contained with well defined functionality that the compiler can analyze ahead of time.
- Most performance sensitive code paths in games are of numeric nature and largely monomorphic.
- Dynamic code paths are usually explicitly placed and the impact of them in a soft typed system is relatively small with modern interpretation techniques.
- That callsite of data allocations (formally 1-expansion as in 1-CFA) by default is sufficient for successfully inferring most performance sensitive code in practice.
- In cases where1-expansion is insufficient optional programmer guided expansions should only be needed for corner cases like container libraries like binary trees.
- Basic container libraries are rarely used in practice as EcmaScript developers mostly use object literals as hash tables and the regular Array type supports vector functionality.
- Undefined semantics can be modified as most computations on the Undefined value are accidental and result in unwanted behaviors anyhow.

Based on the all the reasons and code usage hypotheses the designs of the compiler and runtime system was set.

- The compiler has to be fully ahead of time, this means that optimizations has to be possible to calculate ahead of execution time.
- The compiler will use type inference based on the Cartesian Product Algorithm to infer as much data as possible about the final executable will behave.
- As the full language is impossible to optimize well the compiler operates on is a subset of the EcmaScript standard that relies on some of the hypotheses about code usage specified above.
- The type inference is not required to produce a perfect inference result as actual dynamism is expected in the code.
- A developer should be able to guide the inference system in specific cases where performance really matters.
- As dynamism is expected the compiler and runtime system should be designed so that the penalty of dynamic code is minimal where it does occur.

4.2 Compiler System Design Overview

As shown in Figure 1 the compilation process is divided into a number of major stages. The first stage is parsing and lexing, that takes the source files of a program and turns them into an abstract syntax trees (AST). The second stage is rewriting of the AST in order to simplify the code. The third stage is a transformation of the AST into an internal form with continuation passing style(CPS) characteristics. The fourth stage is an abstract interpretation of this internal form to produce a control flow graph and abstract type information. The fifth stage rewrites the abstract code paths into code paths suitable for execution by a regular cpu. The sixth stage transforms object definitions based on information in the abstract data to real object definitions that can be used during execution. The seventh and final stage produces C code that can then be compiled by a regular C compiler.

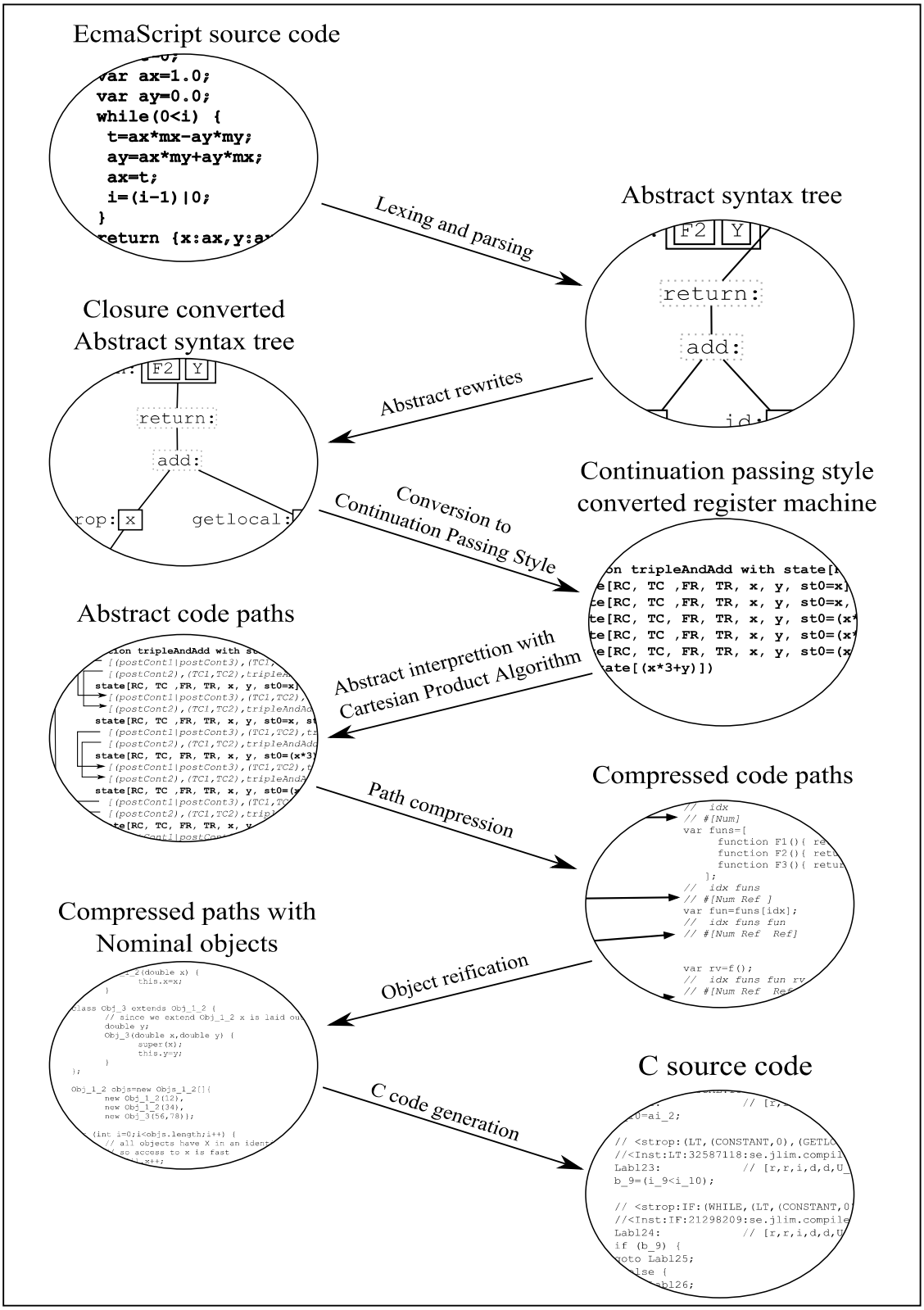

Figure 1: Compilation process overview

4.3 Parsing, Syntactic Analysis and Closure Conversion

The first stage of the compiler is relatively straightforward and doesn't deviate from how most other compilers operate. First a lexer and parser combination designed to handle the standard EcmaScript syntax is used to parse the programs sources files to produce an abstract syntax tree (AST) in memory that is used throughout the rest of the compiler. A simple example of this is shown in Figure 2.

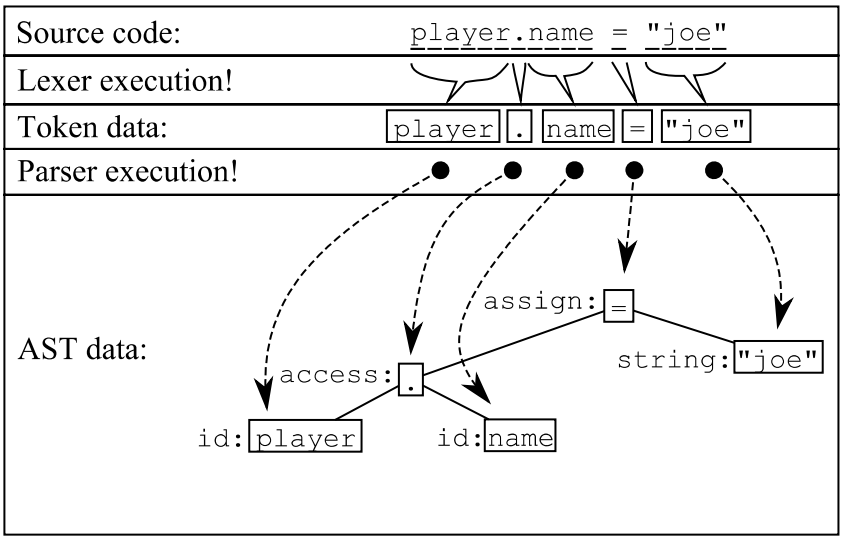

Figure 2: Parsing of source code to abstract syntax tree

A lexer is based on applying lexicographical rules detailing facts such that identifiers always starts with alphabetical characters or _ and can then after the first character be followed by alphanumeric characters. These rules makes it possible to produce state machines or other functions that combine single source code characters into source code tokens. A token could be an identifier (such as *player, name, name30*), a number (0, *1, 30, 10e5, -0.2*), a string (*"joe"*) or some operator such as dot (.) or equals (=).

After the source characters are converted into tokens by the lexer, a parser interprets these tokens in relation to each other to interpret things as facts in an Abstract Syntax Tree. In the figure for example the root tree nodes specifies the assignment operation that was parsed out due to the parser associating the equals (=) token with an assignment operation that requires a target (the *name* property of the *player* object on left side of the tree) and a value (*"joe"* on the right side of the tree).

A quirk of parsing EcmaScript due to regular expressions is that lexers are required to be pull based as the lexicographical rules depends on the current parsing context that makes implementation of the lexer and parser combination slightly more complex than for many other languages where the lexer and parser can be decoupled. The compiler implements the lexer as a finite state machine while the parser is based on the common recursive descent method.

Once parsing is completed a couple of relatively simple syntax analysis and rewrites operations are executed on the AST. This means that these analysis and rewrite operations works on a semantic level very close to the source program as written by the developer.

```
function F(){
       // x stored in implicit
       // environment object
       var x=100;

       // Creates a new function
       return function F2(y){
              // The x access refers
              // implicitly to the
              // environment object.
              return x+y;
       };
}

// Call F to create F2 with it's env.
var f2ref=F();

// Returns 102
f2ref(2); // returns 102
```

Figure 3: Variable environment example

Most importantly a simple environment analysis is done that resolves variable accesses between different functions and scopes. This connects the variables, which the programmer uses to the function environments they belong to as exemplified in Figure 3, in the figure the usage of variable "x" in the inner function is called a non-local variable or more formally a free variable. With environments created and variables attached to them a transform known as closure conversion can be done (Appel 1992, p. 103). This is done by the compiler in 2 stages.

The first stage is done by utilizing the variable locations to identify variables accesses that are fully local to a function like "y" in the figure(i.e. both the definition and all usages are within the same function) from the free variable accesses that occurs when there are references to variables defined in a parent function like as exemplified in the previously mentioned figure with the variable "x".

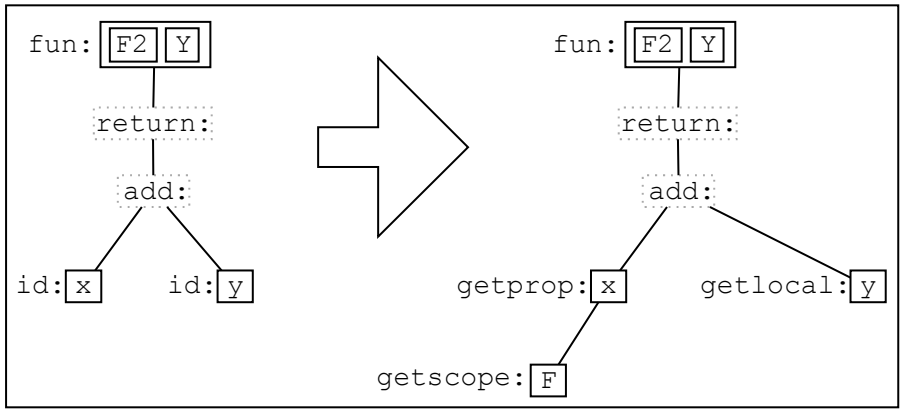

Figure 4: Closure conversion of function F2

With the identification done the compiler executes a rewrite on the AST to remove from the code the free variable accesses, replacing them in the syntax tree with explicit object property accesses that works in the same way as regular object accesses would (free variable accesses would need to implicitly load an environment object before operating upon a variable). This is exemplified in Figure 4 that shows the AST layout of the function body of the F2 function in Figure 3 before and after the rewrite stage, the implicit variable accesses in the function are transformed into unambiguous and explicit get operations.

With the closure conversion done on the AST form that yields internally functionally equivalent code, later stages of the compiler are simplified since all complexities of free variables in lexical scoping are removed and replaced with regular object accesses that the compiler must handle regardless.

4.4 Advanced Flow and Type Analysis in the Compiler

4.4.1 Limits of an Abstract Syntax Tree

Syntactical analysis stages can answer most questions related to type information in explicitly typed languages such as Java, C# and C++ with minor function local analysis needed in the worst cases. Such syntactical analysis works well with walking the hierarchical structure of an abstract syntax tree with a simple recursive function, for a language with dynamic types and first class functions on the other hand there is a need to do extensive global flow analysis to even be able to determine what functions are invoked with what kind of arguments as the functions are passed along as values in the target language rather than as syntactically bound symbols (Shivers 1988). While it's not immediately obvious what flow analysis techniques are referred to in Shivers paper it does not matter as the main point he made in this paper was that control flow in these kinds of languages are dependent on values that are computed during execution. To recover the control flow and type information from the source program, the compiler will execute an abstract interpretation of the program. First the compiler converts the program from the AST to an equivalent of the continuation passing style and secondly the compiler applies an abstract interpretation of program to be compiled using the Cartesian Product Algorithm with abstract values as described in the coming passages.

4.4.2 Conversion to Continuation Passing Style (CPS)

A continuation is a function defined as the rest of the program to be executed (Reynolds 1993). A tail call is such a call within a function that nothing else will happen in the function after call is made that can safely replace the callers stack with the callees. With these two definitions continuation passing style(CPS) is defined as programs, where functions are constructed so that control flow is done by tail-calling simple functions and passing along what to do next as yet another function(the continuation).

While this definition might seem convoluted the requirements for mechanically handling such code is in reality very simple and this format is often actually used in compilers for many optimizations and transformations up until the point that actual primitive machine specific instructions need to be generated (Steele, 1978) (Appel 1992).

Figure 5 shows a very simple example, in it add and mul is used as a function instead of the + and * operators to emphasize the point that all execution is modeled as function calls. The original function in the example is a function that receives 2 arguments, multiplies x by 3, then add y to the return value from the multiplication and finally return the result to an unknown function. In contrast the closure converted function is composed of 3 functions that takes input argument(s), sends them off to an arithmetic function that in turn calls the next

function with the result. While longer overall each separate function in itself is simpler since the only thing it does is call another function with variables, constants and other functions as arguments.

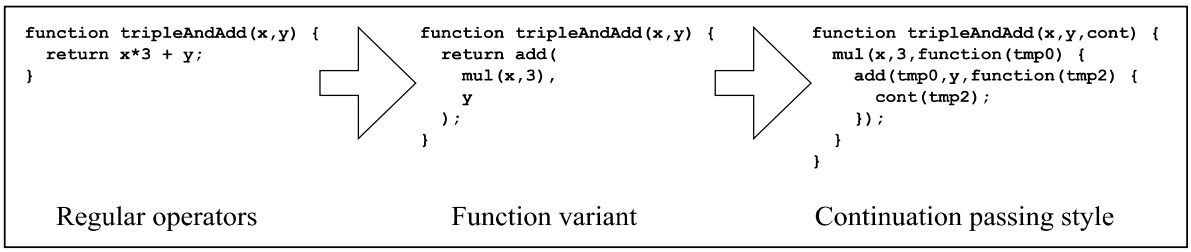

Figure 5: Regular continuation passing style conversion illustrated

Just as with the closure conversion described previously the main purpose of transforming the code into CPS is to remove complex implicit semantics from the compiler internals and replace them with explicit operations. In the example above the main benefit is that there is no implicit returns, rather it is explicit that a function closure is explicitly invoked with the result of each computation. Apart from knowing the return function explicitly continuations can also be used to model the non-local throw/catch semantics of exception handling, and in many ways these can be even harder to analyze than function calls since a throw doesn't just return to the immediately calling function but can be caught by an exception handler in a function that indirectly called the function that throws the exception.

The example of CPS conversion in the figure above introduces the new free variables of y and cont as accessed in the inner functions, and while this is useful to the expository purposes of explaining CPS it contrasts slightly with how the compiler works in practice as it never creates any free variable during the conversion. The compiler instead directly creates a form that reminds more of what Appel(1992, p. 119) describes as callee-save continuation closures but with an infinite register set, the internal structures within the compiler also departs enough from the source notation that examples will be shown in an abstract notation to better reflect implementation realities.

```
function tripleAndAdd with state[RC,TC,FR,TR,x,y]
 state[RC, TC ,FR, TR, x, y, st0=x]=append(state,state.x)
 state[RC, TC ,FR, TR, x, y, st0=x, st1=3]=append(state,3)
 state[RC, TC, FR, TR, x, y, st0=(x*3)]=multiply(state)
 state[RC, TC ,FR, TR, x, y, st0=(x*3), st1=y]=append(state,state.y)
 state[RC, TC, FR, TR, x, y, st0=(x*3+y)]=add(state)
 RC(state[(x*3+y)])
```

Figure 6: Output of conversion to abstract machine

Figure 6 illustrates a form very close to what the compiler actually produces from the tripleAndAdd function that was shown earlier in Figure 5. In the figure the register set is denoted as state, this register set is used to contain all local variables as well as temporaries of an evaluation stack similar to a stack machine. Within the state RC and TC represents the virtual return and throw continuations (the virtual part being due to them only existing during abstract interpretation and being ignored in the final generated code), FR is the function closure reference that might be needed to access environment variables that was extracted during the earlier closure conversion, TR is the JavaScript "this" reference and finally x and y represent the regular function arguments.

In the notation each call and assignment corresponds to an independently executable instruction. The last execution in this example executes RC to emulate a return statement, when executed the result of this function is appended onto the local state at the call instance that invoked the tripleAndAdd function similar to how append, multiply and add worked within the function. Had TC been invoked instead the control would have transferred to the matching try/catch statement(s).

The representation used in the compiler deviates in slight ways from what could be considered pure CPS conversion. In a sense the function local parts could be viewed as a regular control flow graph, yet those flow constructs are not used after abstract interpretation and the compiler also has a relatively coherent handling of "regular" local operation nodes and continuations that follows the blueprints of CPS based compilers. However the main benefit of the local and non-local distinction is that the compiler only needs to handle free variables during abstract interpretation for the virtual return and throw continuations. Thus the compiler has no need of more transforms on the code between abstract interpretation stage and the code generation leaving most of results of the interpretation available to the code generator to do optimizations such as generating fast monomorphic type accesses instead of generating slower generic code.

4.4.2 Abstract Interpretation with the Cartesian Product Algorithm

Abstract interpretation is described as the compiler correctly running all possibly occurring functions and branches of a program with abstract values instead of real values. The differences are due to the fact that the final compiled program will run with data and inputs that are unknown at compile time. So while the exact control flow cannot be known ahead of time for all programs, a compiler can extract a conservative approximation of the control flow that can be used for various purposes by doing abstract interpretation. Exactly how abstract interpretation is implemented always depends on the requirements of the source language and the design of the system and on how much information is needed for the compiler to accomplish the required goals.

Compared to normal execution as done directly by a CPU that deals with basic types such as booleans, numbers, objects, functions and so on, abstract interpretation deals with descriptions and abstractions of those. For example during normal execution exact numbers such as 1, 2, 3.... might be passed to a function as the parameter X, the compiler doing abstract interpretation will use representations such as “anything”, “a number”, “an integer number”, “a number greater or equal to 1”, “a finite set of the numbers 1,2,3,4” or combinations of those. As a corollary to this follows that while normal execution of a statement such as “if (X<3) A else B” will run either A or B since X is known exactly, an abstract interpreter having the abstract values as described will not be able to decide exactly if A or B needs to be run and thus needs to run both. Similar to this, instead of executing on actual objects and functions(functions being objects themselves in EcmaScript) as during normal execution, objects during abstract interpretation are represented as sets of abstract objects that themselves contains possible sets of properties rather than exact properties.

The Figure 7 exemplifies abstract interpretation of a call to the CPS converted tripleAndAdd function presented earlier, the contents of state between each basic computation unit is shown as [Value1,Value2, …, ValueN]. It shows an equivalent of a call from CallSite1 to tripleAndAdd with postCont1 as a continuation representing the return position, a constant Number (2) sent as X and finally an unknown Number sent as Y. One can see how with each instruction the abstract interpretation propagates most values and changes some. By following the execution states it is observable how the constant is propagated and compiler recognizes that the multiply function operated on constant values and produces the new constant 6, the following add instruction however also operates on an unknown quantity and thus produces a final unknown number.

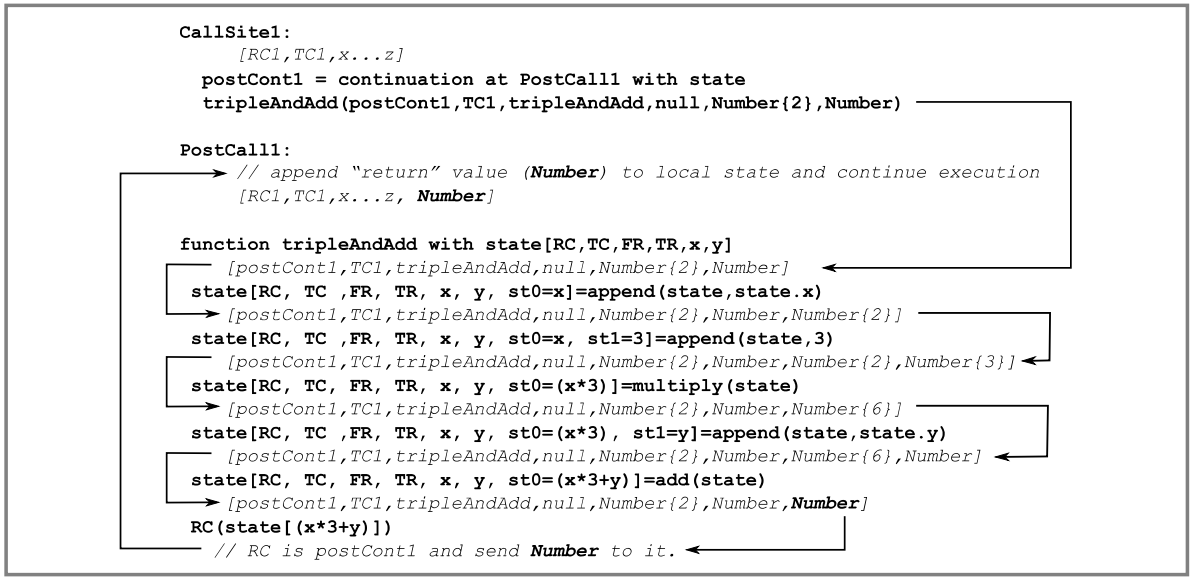

Figure 7: Basic abstract interpretation

Even with a naive and basic abstract interpretation system as in the example above the compiler would be generating optimal code, however EcmaScript has a number of semantics that are troublesome to analyze and one particular example is the semantics of the + operator where the result type can be either a Number or String depending on the input arguments. While it could be possible to represent this as (Number or String) we will use the notation Anything for sets containing multiple basic types since a runtime type test for 2 basic types is usually as expensive as testing for all basic types.

Figure 8 shows how this affect basic implementations by adding another callsite to our tripleAndAdd example, this second callsite is equivalent to the first apart from that the return continuation is postCont2, X is an arbitrary number and Y is a String. Since the two call sites execution merges at the entry of tripleAndAdd the execution state from there on has to be merged. First the continuations are kept as sets of continuations (so that returns and throws can be executed on all relevant targets), secondly in the X argument slot the constant Number{2} is merged with an unknown number to produce another unknown number and finally in the slot of the Y argument the passed Number and String are merged into Anything. Looking at the execution of the function it can be seen that since the X argument is no longer constant the multiplication will produce an unknown number and in the same fashion as Y was Anything and the addition operator operating on Anything will produce a Number or String (equivalent of Anything) then the result of this operation and the entire function will be Anything.

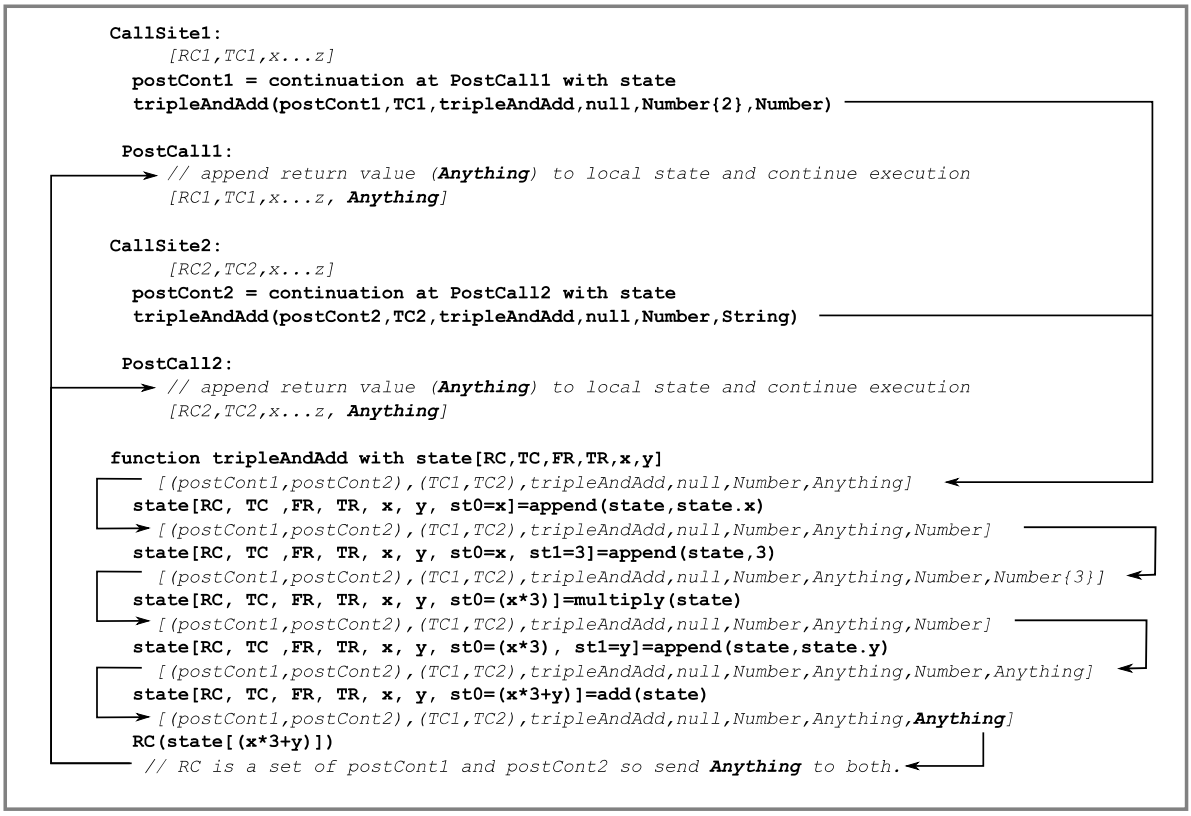

Figure 8: Problematic example with basic abstract interpretation

During normal execution the cost of executing simple arithmetic operations such as additions with runtime type tests due to unknown exact types usually more than halves performance compared to directly executing explicit cpu instructions when exact types are known, even if the penalty exact factor depends highly on actual implementation details and how the target CPU there will invariably be a cost. So the just presented example where Numbers and Strings merged to force Anything operations would have made the tripleAndAdd function significantly slower. Even worse since the output of the function became Anything, these values requiring type tests would propagate beyond the tripleAndAdd function itself from the PostCall1 and PostCall2 positions to potentially unrelated locations in the program slowing down not just this single function but possibly large swaths of the program.

While finding an exact solution to type inference problems is subject to the halting problem a variety of algorithms use different heuristics to find better solutions. All of them will in various fashions duplicate code paths so that the individual generated code paths can run as quickly as possible with a reduced number of type tests required compared to the base scenario. Of these algorithms this compiler is designed around the Cartesian Product Algorithm (CPA).

The Cartesian Product Algorithm originally defined by Agesen is defined simply put as a Cartesian product between all possible types for every argument slot for every function, while

that simple definition would use a very large amount of memory the algorithm also specifies that building the product can be done lazily in a monotonically increasing fashion since the algorithm only needs to consider those new combinations that are actually possible. In contrast with many other algorithms based expansion like n-CFA and IFA, the requirement that the set building is done in a monotonically increasing order is a key detail of the algorithm since it makes it very stable in the sense that no decisions are reversed and once the algorithm has filled in all reachable argument combinations it will terminate.

The compilers implementation of the CPA algorithm works on a per-operation basis (equivalent to functions in CPS terms) since in EcmaScript function calls, node fetches and even operators can produce multiple types the analysis needs to work with a finer granularity than entire EcmaScript level functions. Also objects and functions are given the same precedence as numbers in the compiler during abstract interpretation, so objects from different allocation sites cause disjoint analysis paths since their contents can vary and the functions they target differ. In addition to this the compiler modifies the monotonicity definition slightly to account for details in the EcmaScript language without breaking the stability, the modifications all involve the notion of merging specializations into supersets. This is possible because merged values can never devolve back into a subset and a superset will always accept everything defined in the subset it evolved from during the merge. There are three primary examples of how merging values is done in the compiler.

The first example is continuation values. Since once 2 paths have merged the same path will be followed the compiler works with continuation sets rather than actual positions like with objects and functions. The result of 2 continuation sets meeting is always the union of those sets and by that definition that union will always accept both of the subsets that was used to produce it.

The second example is numeric promotion. Most optimizing EcmaScript runtimes tries to use 32-bit integers if possible since they are usually faster than IEEE 754 double precision floating point numbers that the EcmaScript specification specifies as the number. The compiler also does this where possible but if a Number and Integer value crosses paths the compiler must always merge the result to be a Number since the possible values of double precision floats are a superset of 32-bit integers.

The third example is if the compiler has during abstract interpretation first detected a constant such as Number{0} for an operation (often found as the value for the first iteration of a loop) and then subsequently encounter that same operation with another number the compiler then merges the individual constant numbers into a generic number (the compiler could potentially calculate bounded numbers but that has not been implemented). From that point every time the compiler runs that operation again, any new number, constant or not, will match the existing generic Number definition. In this sense the merging could be seen as a

generalization of infinite sets. This kind of infinite set generalization merge also applies for booleans and strings.

In Figure 9 we again look at the abstract interpretation of the tripleAndAdd function, this time using the Cartesian Product Algorithm adding a third callsite that that sends a constant Integer{2} as the X argument and another constant Number as the Y argument. During abstract interpretation tripleAndAdd is first called from CallSite1 and this produces an initial path since there existed no paths before this. After this CallSite2 is encountered and since Number and String are considered as fully disjoint types during abstract interpretation the compiler produces a new path through the function (the compiler will try to merge the paths if possible but the paths at all positions are disjoint in this example). Lastly the newly added CallSite3 is encountered, this time the compiler detects that while not fully identical the arguments passed here are compatible with the path produced by the earlier call from CallSite1. The compiler now begins to merge the new path with the old path to produce a new path that is a superset of the 2 previous paths. First as per the first merge example a union set is created of the 2 continuations since they will both follow this codepath. Secondly as with the second merge example the Integer constant from CallSite3 to arguments X is promoted to a Number(double precision float), since both constants are now equal in type and value it will remain a constant. Thirdly the Y argument has 2 Numbers but with different constant values and as with the third merge example the superset is all Numbers. After the merge operation at the function entry point the subsequent states produced by CallSite1 will match apart from the Continuation set that needs to be merged for all subsequent operations.

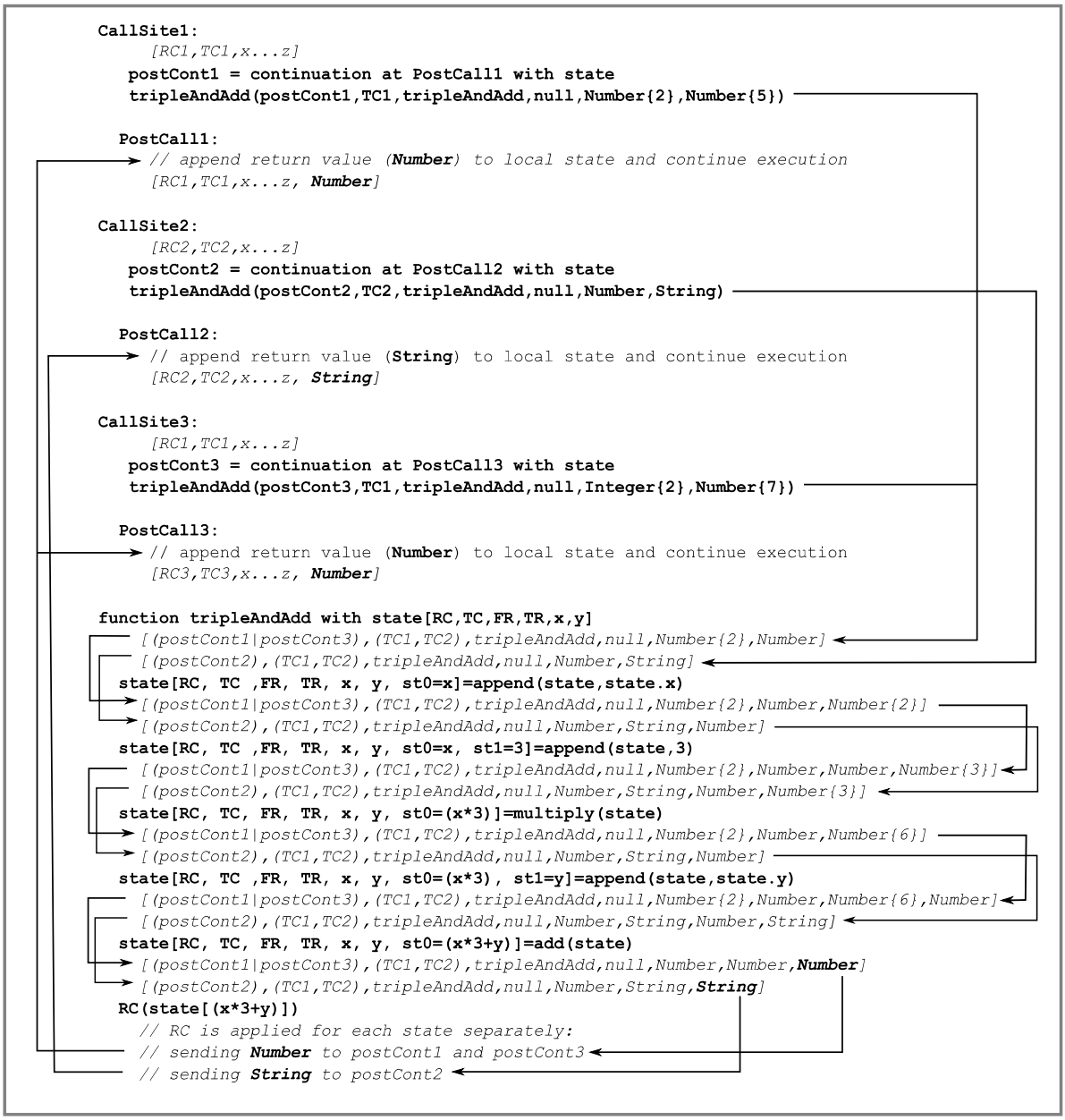

Figure 9: Abstract interpretation with the Cartesian Product Algorithm

Compared to problematic behavior shown in Figure 8 where the naive merging of Numbers and Strings would have produced slow code paths due to runtime tests associated with the catch all Anything type, the abstract interpreter example using the CPA algorithm as shown in Figure 9 analyzed CallSite1 in an equivalent way to the simple case first presented in Figure 7 giving exact type information that could be used to produce optimal code. Also like the simple case no slow generic type would escape the tripleAndAdd function to slow down other areas of the program as in the problematic example with naive interpretation. As a side note, while it might be possible to get even faster code by not merging the Integer and Number paths it would increase analysis time and would require a backend that can detect

overflows of integer additions, something that a pure C backend is incapable of doing without performance penalties.

4.5 Nominalization of Abstract Structural Types

The output produced by the flow analysis is in a very abstract and verbose form. This means that while all data is necessary to correctly analyze the program, the data about the program is magnitudes larger than the data actually needed to produce executable code. So to make the data from the analysis useful for actual code generation two different processes are applied to the resulting data to make code generation feasible and efficient.

The abstract interpretation handles objects in what is called a structural form that only contains information about what properties exist in objects but has no data about how the objects are actually laid out in memory after compilation. While the structural form greatly simplifies abstract interpretation, the information about how all objects are laid out in memory is something that is required by the compiler to produce optimized code

Nominal types are named types as used in languages like C++, Java, C# and so on, usually these types has specific layouts that makes it easy for the compiler to make fast property accesses since the layout information is available to the compiler and ends up as an offset for pointer accesses. Nominal types are also used in hierarchies where the types have a specific layout and all child types have identical layout in memory to their ancestor types for all shared properties. So to generate as efficient code as the above mentioned languages, compilers of dynamic languages has to find ways of mapping the property accesses to accesses on Nominal types with explicit memory layouts.

Other compilers using CFA for analysis build nominal types during the abstract interpretation stage. The Starkiller compiler uses something that could be called instance re-flowing where allocation sites contain the object shape and each time a new property is introduced or generalized the object at the allocation site is changed and reanalyzed.

The Shedskin compiler on the other hand uses the Iterative Flow Analysis method on objects. The IFA method works by first running a simple flow analysis with one live empty object type initially, after this the algorithm checks for incompatible object usage by finding unknown properties or incompatible types of properties as the Number and String usage described in the previous chapter, if any incompatible object usage was detected the live objects in the analysis are split to remedy the incompatible uses and the analysis is run again until no more incompatible uses are found.

While the Shedskin approach can give the highest potential performance, it will fail analysis for a number of programs due to the sacrifices made in terms of compatibility to achieve optimal performance. Starkiller is similar to the compiler described here since both use

allocation sites as the primary object identifier during analysis, however in cases where polymorphic accesses can occur Starkiller seems to fall back to expensive runtime property searches.

4.5.1 Path Compression

The first stage of the nominalization process is to compress the control flow description into simpler more generic terms better suitable for actual code generation. To accurately analyze code the abstract interpretation needs to keep separate paths for each function and object site that passes through a certain state variable to accurately judge their usage or apply all possible destination functions as illustrated on the left side in Figure 10. However, for real code all these cases would generate identical code of passing the value in a reference at the same location as illustrated by the right side in the figure, thus already all such simple cases would end up as identical code in the compiler binary taking unnecessary space if the compiler did not compact the abstract paths.

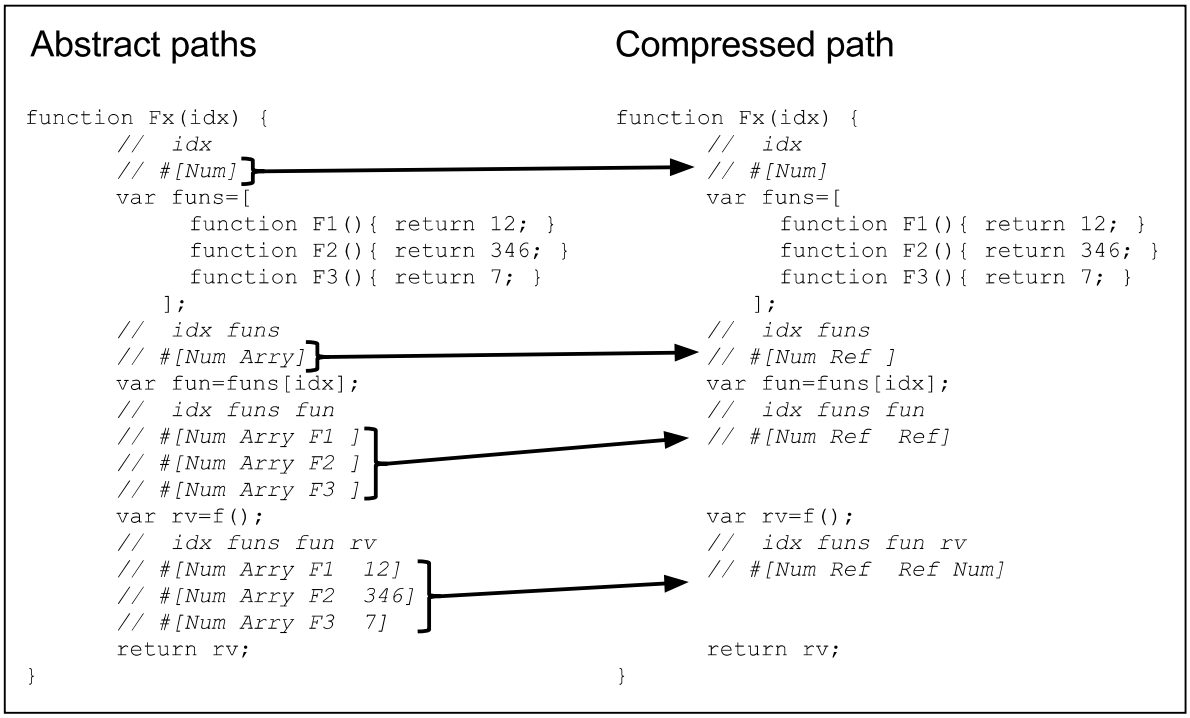

Figure 10: Compression of redundant paths for code generation

While compressing only this trivial case can reduce code size by orders of magnitude, the code size would still correspond more to the abstract analysis size than input code size and produce output binaries with fairly unpredictable sizes to a developer due to the primitive types such as numbers, booleans and references being disjoint in the analysis stage. As a hypothesis set up during the construction of the compiler is that performance critical paths

will end up with very little polymorphism if the developer needs performance by hand tuning code, the compiler can make some assumptions to better match the number of paths to the input code size.

The algorithm implemented to compress the paths iteratively collects all possible execution paths for each built output function and then produces a fixed number of optimistic optimal paths for each operation. The optimal paths would be selected as those that will yield the highest performance by favoring cases where operations using numbers and booleans are executed compared to cases where operations use objects and generic types. After selecting the optimal paths the algorithm then defers all other abstract code paths to a final generic code path that must be capable of handling all possible cases found by the compiler.

If the code always follows the optimal paths selected by the compiler the performance should be close to a theoretical optimum while the worst case scenario of this approach is that the generic path would be active for all code execution but even for the generic path the compiler will generate fairly fast code for many cases due to how programs are usually written. One could theoretically engineer specific cases to slow down the compiled code down to a level similar to interpreters but such cases should in most cases be synthetic in nature.

4.5.2 Object Reification

The second stage of the nominalization process builds real object descriptions in nominal form to be able to capture multiple abstract objects that in reality are equal. Figure 11 first illustrates 3 objects in an array, the first 2 objects have only the property x written to them with an integer value and while they would be separate objects during abstract analysis it is trivial for the compiler to identify them as identical objects and compact the first 2 objects to the same real object type during this reification phase.

```
var objs=[{x:12},{x:34},{x:56,y:78}];



for (var i=0;i<objs.length;i++) {
      objs[i].x++;
}
```

Figure 11: Polymorphism with differing object shapes

The last object in this example however is not identical to the 2 first objects yet all of them are allocated into the same array and will all thus flow into the same code paths. The situation where similar but not identical objects flow into the same code paths is not entirely uncommon and the objects would then with a very high likelihood be used at the same locations in many other places also.

The algorithm used by the compiler to create nominal types analyzes the code and computes confluence scores for property groups based on how many times particular properties are used by different objects at the same locations, the algorithm then picks the property groups with the highest confluence scores to become the basis for creating new nominal types to represent the structural types when generating the actual nominal objects types.

The 3 objects in Figure 11 would produce a property group for their x properties with a high confluence score due to their common increment inside the for loop and the algorithm would due to the high confluence score then place them in the same nominal type. The last object with the additional y property would then introduce another property group for that y property and the algorithm would subsequently create a subtype to hold this y property and move the mapping of the third abstract type to point to this new subtype. In practice this would mean that the compiler could then generate a direct memory access to the field x in all 3 objects in the same location in memory since they all belong to the same hierarchy and hereby speed up execution by not having to query the object about the position of x since all 3 objects would have identical object layouts as far as the property x is concerned. A Java equivalent of how the generated code would look is listed below in Figure 12.

```
class Obj_1_2 {
	double x;
	Obj_1_2(double x) {
		this.x=x;
	}
};
class Obj_3 extends Obj_1_2 {
	// since we extend Obj_1_2 x is laid out identically in memory
	double y;
	Obj_3(double x,double y) {
		super(x);
		this.y=y;
	}
};

Obj_1_2 objs=new Objs_1_2[]{
	new Obj_1_2(12),
	new Obj_1_2(34),
	new Obj_3(56,78)};

for (int i=0;i<objs.length;i++) {
	// all objects have X in an identical layout in memory
	// so access to x is fast
	objs[i].x++;
}
```

Figure 12: Java analogue of the reified object layout

4.6 Code Generation

Writing efficient low level code generators takes a nontrivial amount of time and the main goal of the project was to investigate the feasibility producing efficient code by removing the most expensive high level runtime tests with the help of type inference. As such C code was selected as an intermediary target since there exist very high quality C compilers that has had many man years of work put into producing efficient code with low level optimizations. Selecting C as the target also gives certain platform independence practically for free without having to invest time into separate code generators for the different target architectures. Another big benefit of selecting C as the intermediary language is that writing operating system bindings for various functionality is simplified since C code snippets that is used to do the interfacing can be embedded into the JavaScript code. While C is the current back end code generator most of the heavy lifting to enable optimizations are done in the previously described target independent parts and as such making other backends will be a much smaller job than writing all from scratch again.

The code generator in the compiler is a relatively straightforward system that generates C representations of the Nominal types and builds functions based on the compressed control flow. The generator is also tuned to produce fast code over compact code, so for example all objects have the same relatively big virtual dispatch table(vtable) generated with all unknown properties and invalid function shapes stubbed out with some fault checking code to catch compiler bugs. The big vtable makes generic cases requiring polymorphic property accesses and function calls faster since the compiler can do dispatch on all types without any other runtime type detection than using the vtable supplied by the object itself. However for all cases where the compiler has earlier inferred that for example property accesses are monomorphic (i.e. only one target object type) then it will skip vtable calls and do a direct C property access.

While targeting C as an intermediate language gives good results quickly with relatively little complexity it also provides a couple of drawbacks. The first drawback is that integer overflows checks cannot be done quickly in a portable way. A portable implementation need to produce int64 results from int32 values and this can slow down 32 bit code in addition to the actual tests. Non portable C can rely on compiler intrinsics but would need to retain the portable variants as a fallback. The second drawback is that C provides for no standardized way to walk data in the stack frames of a call chain, this means that to implement exact garbage collection the compiler must keep track of all object references with runtime code instead of being able to rely on a stack walking function to find object references. While a conservative garbage collector could be used there can often be benefits of having exact collectors in terms of latency. The third drawback is that a C based implementation cannot return to different positions in the calling function, such a feature could potentially be used to target different code return positions depending on the computed type instead of passing type information in runtime values.

5 Performance Evaluation

The performance evaluation done is purely focused on benchmarking computational performance. While other metrics could be used to measure how efficiently the compiler and/or runtime is able to analyze and create runnable code, such as the number of successfully removed type tests, garbage collection latency and the like. Those factors are either not relevant (the benchmarks are small enough that the compiler removed 100% of the type tests) or outside the defined scope (garbage collection is a field on it's own and all benchmarks are designed to avoid having memory pressure as an active factor).

5.1 Benchmarking Setup

Correctly benchmarking code performance is a large subject on it's own and can sometimes focus on the wrong things(Ricards 2010) (Ratanaworabhan 2010). However since the produced compiler is not a finished compiler by a far stretch, the benchmarking focuses on a couple of very small micro benchmarks designed to measure general execution cost of particular features and optimizations present in the compiler.

These benchmarks are executed in a variety of configurations and variations to try to give a measurement of the relative performance of the compiler to systems with similar characteristics. Where possible there has been an effort to minimize overhead in timing to give as accurate results as possible but for differences on the order of a magnitude this might have been ignored since the timing overhead would become minor factor in comparison to the system at large.

A mechanical script (using Node.JS) was used to run the benchmarks and collect the data without user intervention during the benchmarking phase. The benchmarks were run 5 times for most tested runtimes and compilers on each platform. An exception to this was made for the very slow interpreters on the ARM platforms that only had 2 runs each, however this does not significantly affect the outcome as the interpreters with only 2 runs were more than a magnitude slower and thus the conclusions drawn due to slightly lower accuracy for the slow tests are unaffected. After running the benchmarks an average of the running times was taken and used for the presented results.

5.1.1 Benchmarked Factors

There is three major factors that play a part in determining the performance of an optimized EcmaScript environment compared to a basic interpreter. They are presented in increased order of compiler sophistication.

The first is interpretation overhead itself of having code to determine the actual operation to be executed. An interpreter needs to find the next operation code, go to the code, find the operation parameters and finally execute the actual needed code. Compiled code on the other hand only needs to do the last part of actually executing the actual operation code, in addition the code can also be tailored to the parameters instead of having to be generic in nature. A subset of this problem is the overhead or limitations of the selected target language (such as C or JVM bytecode).

The second is dynamic typing in terms of varying basic types requiring operators to be executed in different ways. As an example if 2 values are known to be numbers the compiler can generate a native machine add instruction. If the type is not known then a short additional instruction sequence is needed to determine the type, branch and then execute the actual operation for the specific type.

The third and final factor is dynamic property accesses due to hard to predict object shapes, if object shapes are known then property accesses are just indexed loads and stores. On the other hand unknown property accesses can be quite complicated, all depending on the memory model used. The compiler with ahead of time knowledge optimizes object shapes, JIT compilers on the other hand without ahead of time knowledge usually rely on various inline caching schemes. While not implemented the compiler could also be adapted to utilize inline caching for particular cases if the ahead of time analysis is unable to predict the object shape.

Giving an exact overhead of all these factors is impossible as they always depends on how these features are implemented but in general since multiple instructions are often required to do the work of individual instructions in optimal cases the penalty is to be expected to be at least 2-3x for each factor but often worse.

5.1.2 Used Benchmarks

There is a total of 7 benchmarks divided into three sets of functionally similar examples differing mostly in details used to identify the impact of individual performance factors.

The first set of benchmarks (*001_fib* and *002_fibco*) are variations of a regular recursive fibonacci calculation function. The fibonacci micro benchmarks stresses function call overhead and performance of arithmetic operations without involving objects or the garbage collector and can be used to get a baseline of how fast code a compiler can produce in ideal cases. The fibonacci function is timed with a parameter of 35 leading to roughly 30 million function calls, 30 million comparisons, 30 million subtractions and 15 million additions. *001_fib* is a straight application of the mathematical formulation while *002_fibco* is written with integer coercions, the straight application in *001_fib* gives a EcmaScript environment no hints about how to handle numbers while the coercions in *002_fibco* forces all numbers in the

computation to be integers helping compilers and runtimes to produce better code if tuned for this characteristic (the Asm.JS subset of EcmaScript relies on these coercions for the validity of the subset model).

The second set of benchmarks (*003_cplx*,*004_cplxo* and *005_cplxpo*) are variations of computations simulating complex unit number multiplications, these benchmarks has no function call overhead and all operates on floating point numbers. All of them simulate rotating a complex unit vector by multiplying it with another complex unit vector a set number of times (100 million times for *003_cplx*, 50 million times for *004_cplxo* and 10 million times for *005_cplxpo*). The main difference between these benchmarks however is what they operate on. The first in the set, *003_cplx*, does all arithmetic on local variables and has no overhead apart from the arithmetic operations themselves and is an even better indicator than the fibonacci sample on how much an impact dynamic type tests has on arithmetic operation performance since it lacks any function call overhead. The second in the set, *004_cplxo*, is almost identical to the first one but stores the complex numbers in similar objects rather than in local variables and thus indicates how much predictable objects accesses impacts performance in addition to the dynamic typing (or removal thereof). The last one in the set, *005_cplxpo*, is almost identical to *004_cplxo* but puts more strain on the runtimes by sending in differently shaped accumulator objects to test the performance impact of polymorphic objects.

The last set of benchmarks (*006_array*, *007_arrayco*) contains a numeric summation function run as a loop, this benchmark set like the complex set has no function call overhead and instead tests the efficiency of array storage and accesses to them. Arithmetic operations also play a large role in this benchmark but array accesses takes the majority of the performance impact if the array operations themselves are slow. Like in the fibonacci benchmarks the difference between *006_array* and *007_arrayco* is the presence of integer coercions, that in this case can plays a more significant role since the number being coerced is also used as the array index

All the benchmarks were specifically designed to minimize the impact of JIT compilation pauses and garbage collection collection pauses by doing any needed setup and then running five warmup iterations of the benchmark with identical parameters before starting the clock when timing the functions.

5.1.2 Compared Systems

The compared systems are listed here and the identifying abbreviations used in the benchmark charts are introduced here as a reference.

The compiler described in this thesis is compared twice for each benchmark, once with property accesses inlined and once without inlining enabled. This is to gauge the general

performance impact of fast property accesses compared to dynamic property accesses. In the charts these are noted as (*cm_in*) for the inlined variants and (*cm_no*) for the variants with no inlining.

The three major open source JavaScript runtimes with built in JIT compilers, V8, Spidermonkey and JavaScriptCore, were all benchmarked to give both information on theoretical high end performance and the relative performance of interpreters.

The V8 runtime that drives Chrome and Node.JS has been benchmarked with node 0.12 running with baseline compiler only (*v8no_b*) and also with the optimizing crankshaft compiler enabled (*v8no_c*). On one platform 2 different versions of V8 was compared instead due to some performance regression when testing with a newer Node.JS environment, the tested versions were those V8 versions bundled with Node.JS v0.6 (*v8n6*) and Node.JS v0.12 (*v8n12*).

The Spidermonkey runtime used in Firefox is also included in the tests, the Spidermonkey runtime is divided into 3 tiers with increasing performance. First comes the interpreter, normally used to run code that is seldomly executed like initialization scripts (*sp_in*), for code that is executed a few times more the second tier is a baseline compiler(*sp_ba*) similar to the baseline compiler found in V8 and finally the 3rd tier is called Ionmonkey and is the compiler level with most optimizations enabled(*sp_fa*).

The third major open source JavaScript runtime JavaScriptCore not tested on all platforms due to difficulties of building it. Like Spidermonkey it has 3 performance tiers, an interpreter(*jsc_i*), a baseline compiler(*jsc_j)* and an optimizing DFG compiler (*jsc_d*). A fourth tier called FTL is under development but was not tested due to the building problems mentioned.

Three other less known open source JavaScript runtimes was also tested. The old Java based Rhino(*rhino*) runtime and the new Java8 based Nashorn (*nasho*) runtime were tested. The Java based runtimes are included in the testing since they generate JVM bytecode that has similar restrictions on the generated code as a C backend has and provides some hints on how well dynamic code can perform with those kinds of restrictions. And finally the Duktape(*dukt*) interpreter was tested since it has recently become a popular runtime for embedding.

In addition to the public runtimes there was a couple of special tests added. First a C variation of some of the tests were done (*ccref*). The separate C benchmarking is done to give a bound of how roughly how fast it is theoretically possible for the benchmark to run without any JavaScript overheads. Additionally a secondary simple EcmaScript to C++ translator(*cmref*) was developed in addition to the compiler that has been described. While the public JavaScript runtimes might have extra overhead due to interpretation in addition to the

dynamic type tests this secondary EcmaScript to C++ translator gives a good isolated view of the overhead produced by dynamic runtime type tests on the two benchmarks it was able to compile.

5.1.3 Compiler and Machine Variations

The choice of C as an intermediate language means that the runtime performance of the generated code is dependent on the efficiency of C compiler used to compile the code to the native machine, beside C compiler differences the characteristics of the target architecture and even individual cpu can also impact the performance of the produced code.

The compiler selection has varied a bit depending the available platforms but in general the relative performances has not varied too much and as such and the trends has been consistent enough that shallow conclusions can be drawn from the results.

The main part of testing has been done on an Intel laptop (Core i3-2330M CPU at 2.2 ghz) running Windows with the programs compiled by Visual C++ 2013 running in 32bit mode. While GCC can in some cases generate faster code, Visual C++ was used for the default comparison numbers due to more consistent numbers and the fact that Visual C++ is the default compiler for most windows projects due to better availability of API headers.

Secondary testing machines are the Raspberry PI Model B (first model) running Raspbian and a Raspberry PI Model B2 also running Raspbian. The older Raspberry PI B with a 700 mhz ARM11 cpu is comparable to older and cheaper low end mobile phones in 2015 while the newer Raspberry PI B2 with a 900 mhz Cortex cpu should be more comparable to mobile phones popular with consumers in 2015. The code tested on the Raspberry machines was compiled with the Raspbian bundled GCC compiler (version 4.6.3-14).

5.1.4 Benchmarking Variability

To get stable results the Raspberry PI B2 model had all but one core disabled and the Linux kernel “performance governor” was enabled to avoid cpu throttling variations. The older Raspberry PI and the Intel laptop provided relatively stable results when most concurrent processes were closed down. While effort was made to get stable data, anomalies cannot be ruled out as all benchmarks was run on multitasking operating systems.

For example, in one instance 2 of the mechanically produced numbers was removed for one compiler since they were unreasonably high indicating a concurrent process being active (*clang*).
The *cmref* numbers for the *003_cplx* benchmark on the win32 platform were also added manually later from a secondary run since the initial run was made without them, the

numbers should be correct in terms of relative performance as the same preparations were made for running both tests.

Another example of variability is how the C reference code ended up slower than the result of the compiler output on *003_cplx* on the win32 test, closer inspection puts the most probable cause of this as the *printf* statements being inside the timing block rather than outside in the C code since more or less consistent 10ms differences in the timing was detected in all the C samples on multiple platforms indicating that context switching latencies could be the culprit. The EcmaScript samples do not share this timing flaw and the differentials does not significantly affect the relative performance measurements apart from in the *003_cplx* sample.

On inspection of the win32 results an error of 1.6% was calculated. This was done by first dividing the mean difference of each benchmark target value set by the mean value of the same set to get a benchmark target specific error, then a mean of those errors was taken. In general the faster runtimes had far smaller errors but the mean is larger due to the slower runtimes having very variable running times.

5.2 Benchmarking Results

The full results of the benchmarking runs are available in the benchmark results appendix (Appendix C), what follows are a number of charts to summarize the results of the various benchmarks run. All chart numbers in this chapter refers to running time in milliseconds and lower results are better due to higher performance leading to reduced running time.

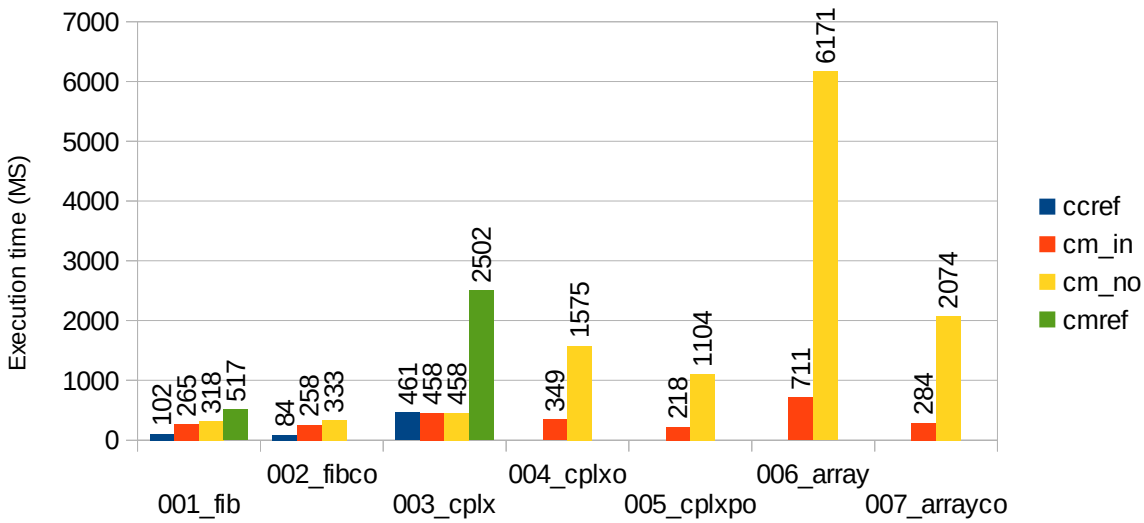

Figure 13: Win32: compiled code execution times (lower is better)

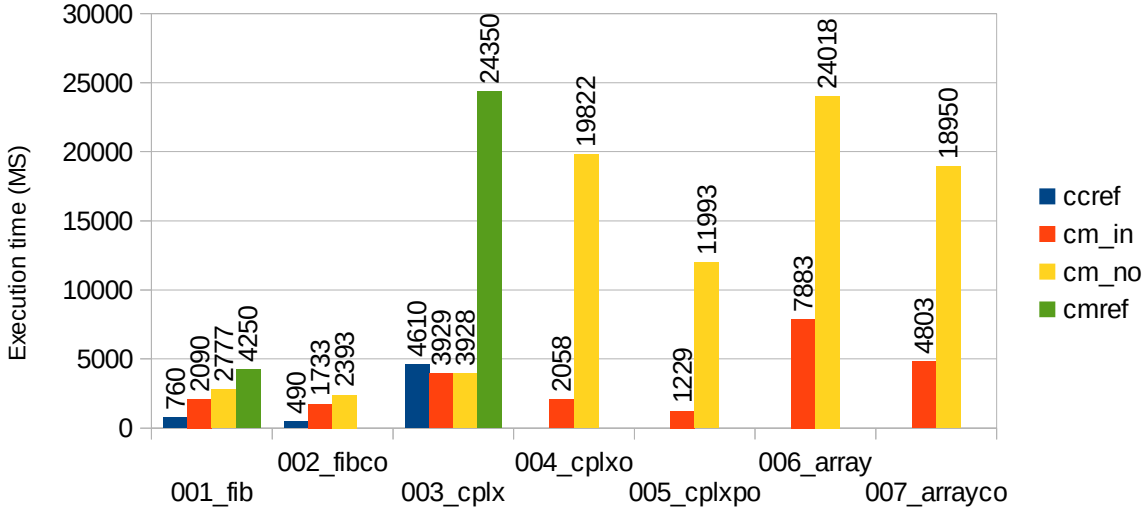

Figure 14: Raspberry Pi B2: compiled code execution times (lower is better)

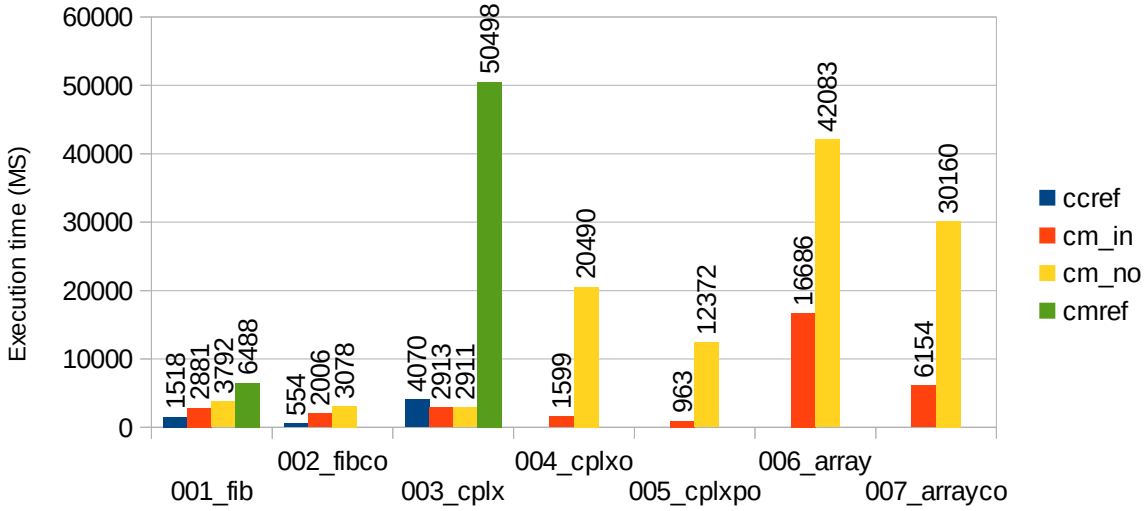

Figure 15: Raspberry Pi B1: compiled code execution times (lower is better)

The figures 13, 15 and 14 shows the performance of the code generated by the compiler with(*cm_in*) and without(*cm_no*) inlining enabled on the different benchmarks. In addition some reference numbers for clean C samples (*ccref*) and the secondary simpler compiler (*cmref*) are included to discern overheads leading to lower performance.

003_cplx is designed to only measure numeric computation performance and has no function call or property access overhead. On this benchmark the main compiler produces code has a running time that is more or less identical to the C reference sample as the compiler is able to infer numeric operations. The simple reference compiler (*cmref*) however only does simple local expression type deductions and is thus only able to remove a few type tests where the output is guaranteed to be numeric, otherwise resorting to expensive runtime dynamic typing

operations. The penalty for the dynamic type test differ significantly between platforms, from being on the magnitude of roughly 5 times longer execution time on the modern Intel laptop, slightly higher on the Raspberry Pi B2 to a staggering 17 times longer execution time on the older Raspberry Pi B1. The exact cause of the differences isn't clear but in general it seems that more advanced cpu's designs somehow handles dynamic type tests better.

The benchmarks *004_cplxo* and *005_cplxpo* are almost identical to *003_cplx* but operates on values stored in objects instead of local variables. While *003_cplx* correctly showed identical performance for the compiler regardless of the property inlining flag, the *004_cplxo* and *005_cplxpo* benchmarks shows a dramatically longer running time with inlining disabled. While the *006_array* and *007_arrayco* benchmarks do different computations by summing array values instead of complex rotations, the running time difference due to inlining can also be clearly seen in these benchmarks.

While the *001_fib* and *002_fibco* benchmarks do show some performance differences due to inlining and dynamic typing those differences are much smaller than in the other benchmarks, instead these benchmarks shows a not insignificant gap in the running time to the pure C runtime samples. On inspecting the code generated by the compiler compared to the clean C samples two things stood out. First is the shadow stack that exists to allow exact garbage collection, while this could be removed by opting for a conservative collector it could also be viewed as cheating as a full blow runtime will benefit greatly in terms of reduced runtime latencies by having accurate garbage collection semantics. The second thing is that method dispatching is always done dynamically through object references in the syntactic closures. This dispatching is in turn visible in three ways in the code, first by the management of the closure objects, secondly by the property accessors and thirdly by the method dispatch itself. The property accessors is what makes the inlined benchmarks faster than those without inlining. The dispatching and closure management could potentially be optimized away but those optimizations are not implemented in the compiler.

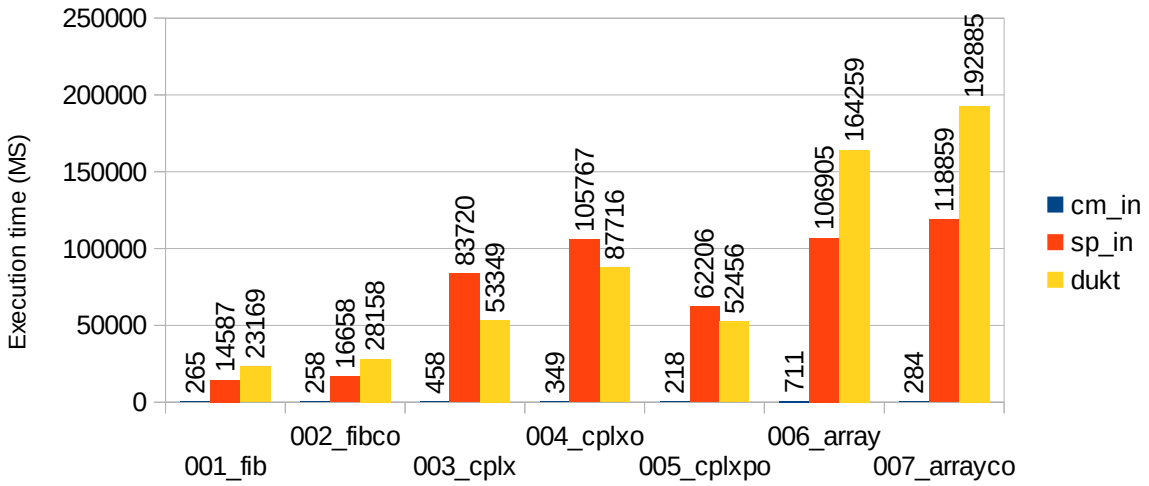

Figure 16: Win32: execution time in comparison to interpreting runtimes (lower is better)

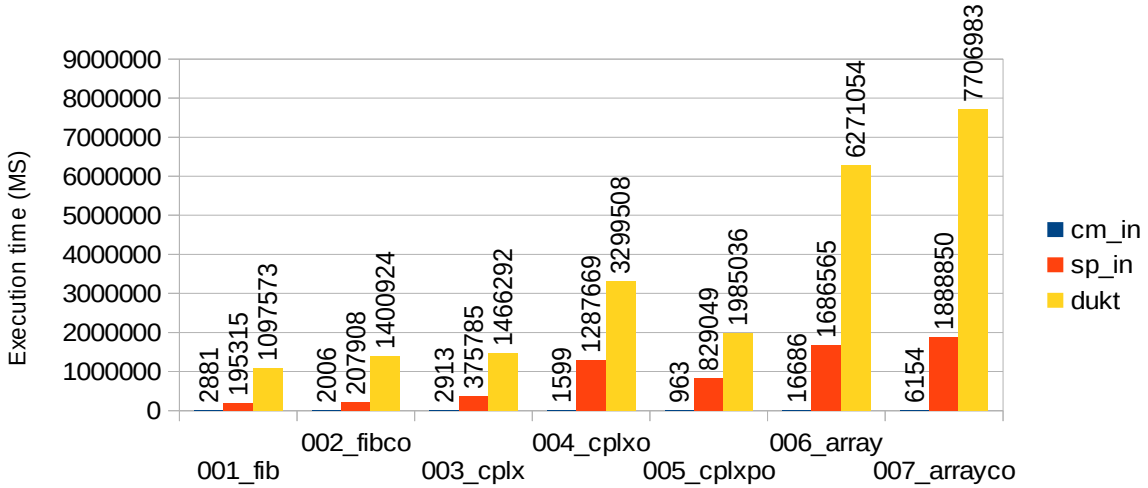

Figure 17: Raspberry Pi B1: execution time in comparison to interpreting runtimes (lower is better)

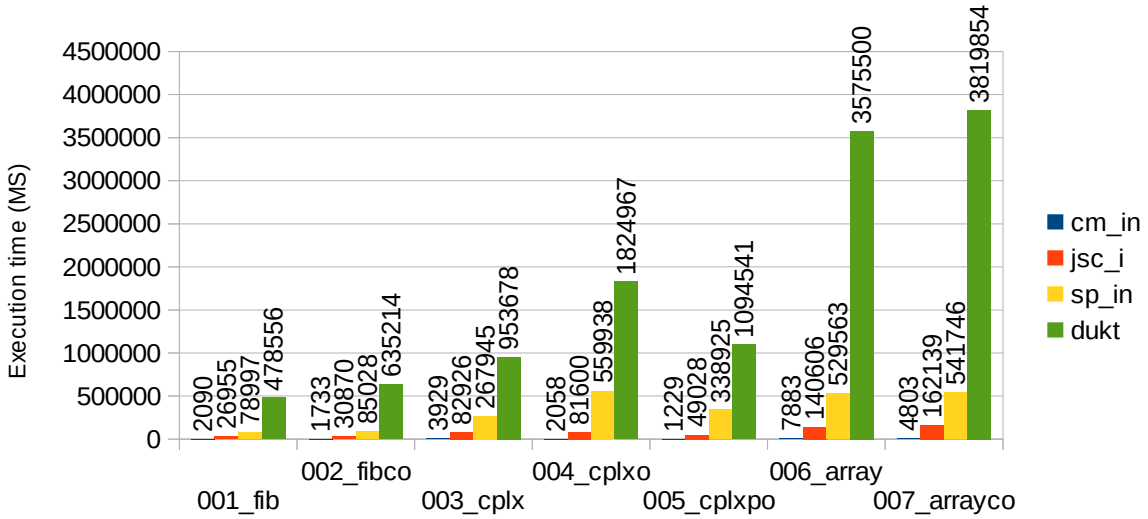

Figure 18: Raspberry Pi B2: execution time in comparison to interpreting runtimes (lower is better)

The compiler is targeted at compiling code for environments that for various reasons cannot use JIT compilers and are thus restricted to precompiled or interpreted code. Figures 16, 17 and 18 shows the results of the code generated by the compiler compared to interpreters. The performance for the code generated by the compiler is roughly a factor from 10 times to 500 times faster compared to the interpreters. If the call heavy fibonacci benchmarks are omitted the compiler comes out even more ahead at roughly 20x compared to best interpreter that was the assembly optimized JSC interpreter.

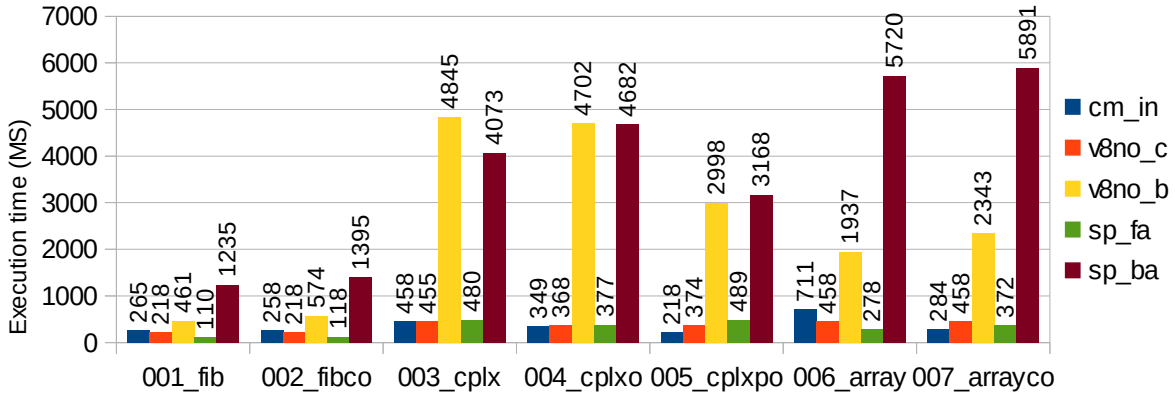

Figure 19: Win32: execution time compared to JIT compilers (lower is better)

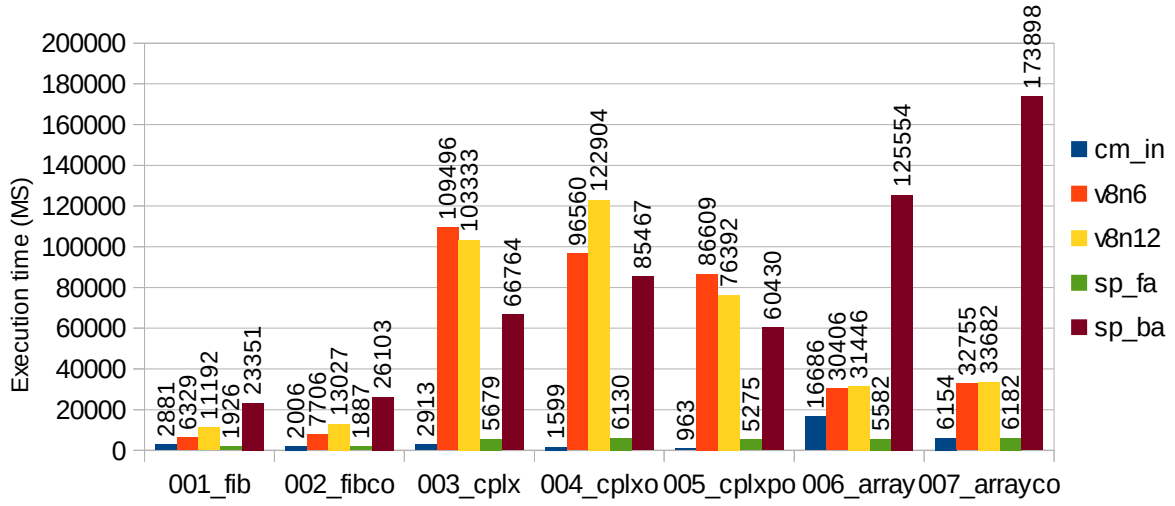

Figure 20: Raspberry Pi B1: execution time in comparison to JIT compilers (lower is better)

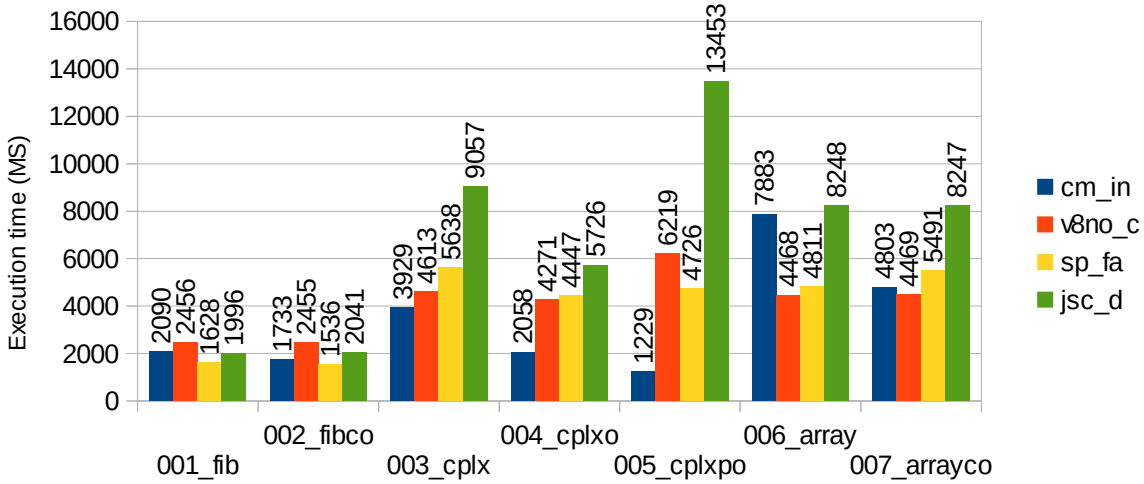

Figure 21: Raspberry Pi B2: execution time in comparison to JIT compilers (lower is better)

Figures 19, 20 and 21 shows the performance of the code generated by the compiler in comparison to open source runtimes with baseline JIT compilers and optimized JIT compilers enabled. In this high performance context it can be seen that the code produced by the compiler is performing roughly on par with the results produced by the optimized open source runtimes, worse in some cases but better in other cases. While the compiler usually fell behind in the benchmarks without coercions, the optimizing runtimes on the other hand had roughly identical results despite the presence of type coercions since the JIT runtimes handle numbers as integers with promotion to double precision floats only after overflow checking.

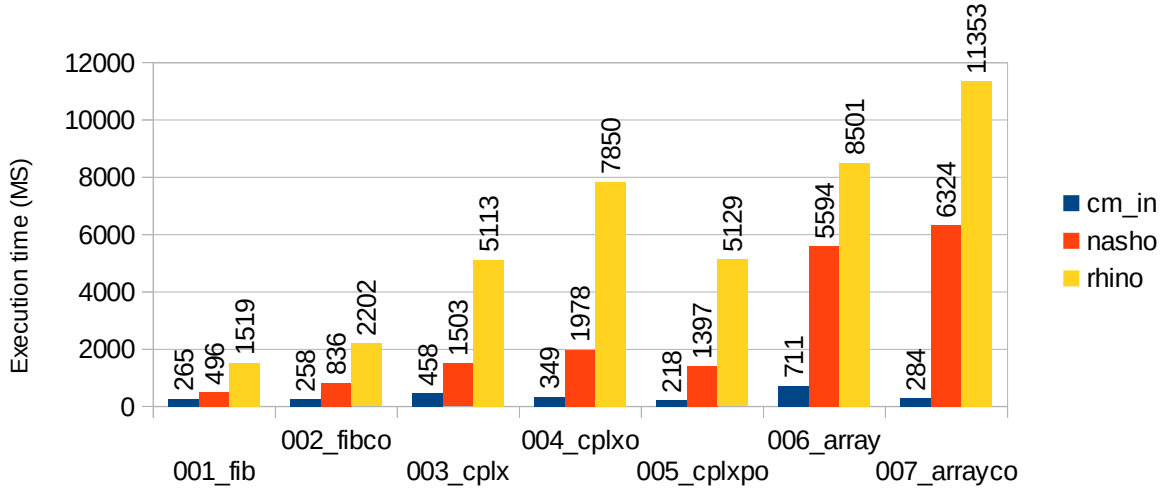

Figure 22: Win32: execution time compared to JVM runtimes (lower is better)

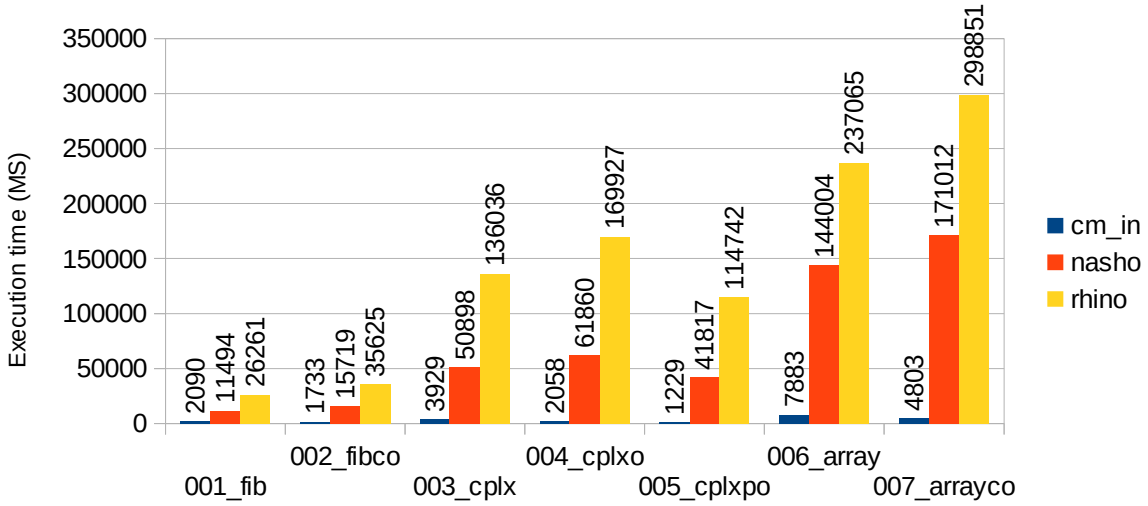

Figure 23: Raspberry Pi B2: execution time compared to JVM runtimes (lower is better)

Figures 22 and 23 shows the performance of the compiler compared to JVM based runtimes, while the JVM runtimes are at a disadvantage since they are built for full compatibility they share some of the limitations with the compiler when targeting C code. The shared limitations

of C and JVM backends shows of how much performance can be increased by a compiler inferring types ahead of time.

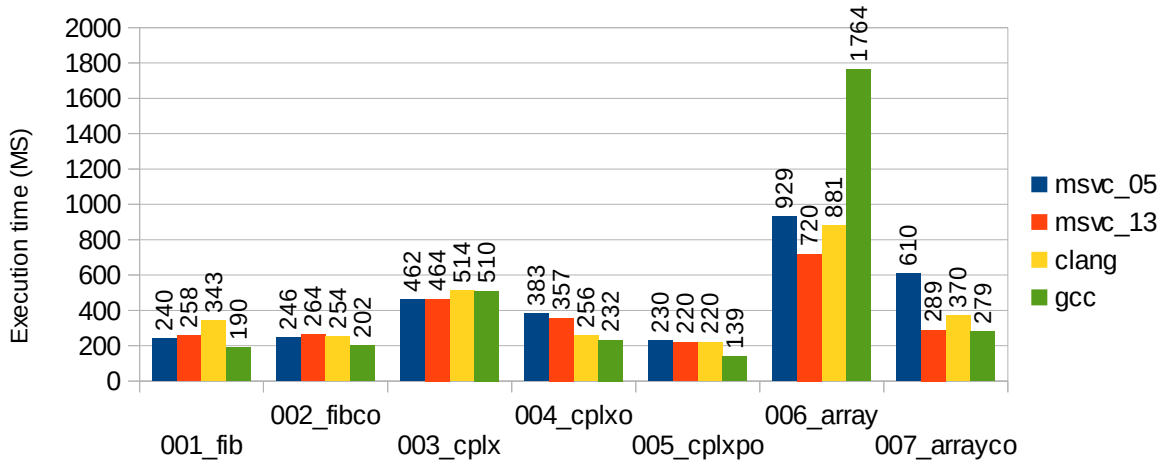

Figure 24: Win32: execution time differences with different C compiler backends (lower is better)

Figure 24 shows the relative performance of using different C compiler backends to compile the C code generated by the EcmaScript compiler into machine code. On win32 the Microsoft Visual C++ compiler from Visual Studio 2005 and 2013 (compiler versions 14 and 18), Clang 3.6.0 and GCC 4.9.2 were tested. The tests showed some mixed results with GCC in general seeming to be edging out Clang and the 2013 Microsoft compiler apart from on one benchmark where some problem made GCC perform much worse than the other compilers. That particular benchmark also showed problematic performance with the Microsoft compilers if one compiler switch was omitted (/QIfist). This compiler switch forces the rounding mode of double to integer conversions to a particular value and exists due to the windows C ABI not guaranteeing that the cpu's rounding flags are properly set at all times, it is not far fetched to believe that this anomaly in the GCC numbers are due to the same problem.

6 Language Subset Evaluation

6.1 Feature Questionnaire

Since the compiler makes certain deviations from the EcmaScript standard a questionnaire was created and posted on various public forums to gather information on if the assumptions made during compiler design would hold up compared to how developers write EcmaScript in reality.

The questions in the questionnaire itself are listed in Appendix D but a brief summary is provided here. The questionnaire starts with a few basic questions about what environments the respondents has experience with. The second set of questions goes into details about the respondents usage of dynamic code with respect to eval and modules to gauge if the compromises and workarounds possible with the compiler are enough to satisfy the developers real world usage of JavaScript. The third set of questions concerns the details of how the compiler deviates from the standard when it comes to handling undefined variables. The fourth set is a single question regarding the optimizations of the compiler in terms of object shaping and their effect on property lifetimes. The fifth set is a question regarding handling of numerical optimizations and how the convenience and compatibility concerns impacts programmers. The sixth and final set is yet another singular question this detailing how developers use regular EcmaScript objects as hashmaps would be used in languages like C++ and Java, this question is highly relevant since the usage is believed to be common while complicating optimizations.

6.2 Questionnaire Results

20 responses were gathered by posting the questionnaire on various forums and social media. Some respondents also reported having some problems understanding the questions due to their technical nature.

While the response set is relatively small and shouldn’t be used to draw definite conclusions, especially as some of the questions might not have been correctly understood, the responses still give a preliminary indicator on the usefulness of the subset defined for the compiler.

17 of the respondents mainly used normal JavaScript, 2 of the respondents used TypeScript and 1 response was given by a developer mainly using Haxe but reporting on his JavaScript experiences. On top of using the various browsers 8 out of the 20 respondents had used some form of mobile porting toolkit in the past.

A clear majority, 16 out of 20 respondents did not use eval in their code. Half of the respondents used other dynamic code loading mechanisms. An anomaly occurred in the

question about developers being able to replace eval with other mechanisms as more respondents answered the question than those who had indicated that they did use eval, possibly due to the order of the questions or poor phrasing. Never the less of those that HAD answered the eval question positively half of those developers indicated that they could replace eval with something else, by counting that would mean that 18 out of 20 respondents indicating that they are able to easily work in the absence of eval.

A little over half of the developers, 11 out of 20 answered that their code should be unaffected to changes to the behavior of undefined. When questioned on the details of the changed behavior that answer did not correlate fully to them expecting their code to behave identically, the number of developers who did not seem to depend on undefined did however remain the same. However most of the developers, 14 out of 20, seemed to be in favor of making the compiler fault on undefined usages due to the error prone semantics introduced by the behavior of undefined that can create hard to find bugs at times

Half of the developers (10 out of 20) thought their code would behave differently if fields existed ahead of time.

Most (13 of 20) of the developers claimed that their code would not be affected by integer overflows, ie letting integers overflow instead of automatically be promoted to double precision floating point values. However when asked about if they would want the compiler to default to a similar mode of operation only 1 out of 20 developers answered positively. 7 of the 20 respondents would have preferred to have a manually enabled mode that disabled integer overflows while 10 of the 20 respondents preferred to use Asm.JS style type coercions as tested in the *002_fibco* and *007_arrayco* benchmarks. 2 of the respondents did not know or seemed to misunderstand the question.

A majority with 16 out of 20 respondents used object literals exclusively for dynamic mappings, 1 used containers and 2 used both object literals and containers. 1 of the responses was “random” that probably meant both but as no clarification was given it.

7 Discussion

The main motivation for constructing the compiler was to build a compiler that would provide a usable high performance environment for game developers building games in JavaScript to enable production of more advanced without being limited by the performance of interpreters. While the compiler is incomplete the runtime performance for the initial well typed benchmarks shows great promise with performance exceeding what interpreters are capable of by more than an order of magnitude even in the worst cases. In fact the performance of the compiler is roughly on par with contemporary leading open source optimizing JIT compilers in most cases, being better or worse depending on the particular benchmark. Compared to JIT compilers the full program analysis employed in the compiler has both benefits and drawbacks.

The main benefit as was shown by the performance wins in the *005_cplxpo* benchmark comes from the fact that the compiler with ahead of time knowledge of properties in an object is able to produce a fixed object layout, JIT compilers on the other hand have to rely on polymorphic inline caches that perform increasingly bad as polymorphism increases. Now the compiler cannot handle an arbitrary level of polymorphism without also suffering but the heuristics does seem to provide some mitigation unavailable to a JIT compiler as shown in the *005_cplxpo* benchmark.

There are however no denying that there are drawbacks compared to a JIT compiler. Full program analysis is not free, the Shedskin python compiler author reports a 2 minute compilation time for a 3000 line program(Dufour 2011). The Shedskin compiler does use a slightly more expensive algorithm than the compiler described in this text but with ever increasing project sizes analysis time and complexity has to be kept in check.

Furthermore certain concessions exist in the performance of generated code in the compiler to keep down analysis complexity, the main one being that the compiler treats most numeric operations with automatic promotions to double precision numbers. This is due to the fact that for the compiler to handle integer overflows as promotions as a JIT compiler would, every numeric operator instance would double the complexity of the analysis for the current method and spill over some complexity increases into other methods as well.

To put this into context, 5 independent variables being operated on at the start of a method would create a 32 times increase in analysis complexity for the rest of the method. In the benchmarks the performance penalty of using doubles in most places instead of integers can be viewed by comparing the runtime difference of the *006_array* and *007_arrayco* benchmarks, where the index being converted from a double to integer makes the compiler handle the first benchmark much worse while being on par with the optimizing compilers on the second one. To some degree this effect can also be seen in the *001_fib* and 002*_fibco*

benchmarks even if the performance penalties are masked to some degree by the function call overhead. These performance penalties reflects the results that Agesen and Hölze(1995) had in their comparison of static and JIT compilation of the Self system.

Future development of the compiler should first and foremost be directed at analyzing bigger programs to give a better understanding of how expensive the type inference algorithm is. Despite the compiler being designed to avoid pathological cases during analysis a better investigation should be done once bigger programs fed to the system to investigate if there are common cases and patterns that would make the code impossible to analyze for the compiler. Once better analysis of the actual analysis complexity has been done, revisiting integer number optimizations without the presence of type coercions should be investigated more carefully in real world settings.

The usage of C as an intermediate target has worked very well when developing the compiler by providing stable low level code generation facilities and thus enabling the compiler focus to be on high level optimizations. Additionally the C code generated by the compiler has been compiled to native programs and executed on Arm, X86-64 and X86-32 with hardly any extra work outside of a few macros and #ifdef's to avoid deprecated header files (The C *alloca* function) and operating system timekeeping functionality. Some of the potential drawbacks of C as an intermediate target was discussed in the code generation chapter. While the integer overflow promotion was partially avoided to keep down analysis complexity in the compiler, not having a standardized support in C compilers for overflow detection during integer operations was another factor in the decision to not implement this. The lack of stack frame walking for C compilers manifested itself in a significant function call overhead, had this been available the shadow stack maintenance in the generated code could have been omitted. Manual editing on the code generated by fibonacci function was performed and showed that roughly 20% of the execution time could have been removed by not maintaining the shadow stack, while 20% is a significant number one has to remember that fibonacci is a very call heavy benchmark that doesn't necessarily reflect real world code.

As a potential replacement for the C backend the LLVM documentation was investigated. The LLVM IR has extensions for handling the above mentioned integer overflow code and LLVM also contains some standardized garbage collection(GC) support routines (LLVM 2015). With these points resolved LLVM seems like a worthwhile target, however in practice these points might be moot. Firstly the integer overflow extensions available in LLVM are also available to C programs with the Clang compiler (newer GCC versions also seems to support the same functionality in intrinsics). Secondly earlier versions of the LLVM GC support also maintains a shadow stack just like the compiler does manually now, thus the performance benefits might not be as big as hoped. Upcoming versions of LLVM however seems to have a new stackmap system in development but this is not yet finalized.

With the compiler hitting the performance targets by nearing the performance of the optimizing JIT compilers and thereby improving upon interpreters, the most important question left is if the subset used by the compiler is useful to programmers. While the compiler was not completed to such a degree to answer all the hypotheses about actual code usage the questionnaire did fill in most of the blanks that were not directly performance related but rather dependent on actual language usage.

Comparing the hypothesis about eval usage and dynamic code loading with the questionnaire results proved mostly correct, while some developers did use eval half of them could move to other methods to load dynamic code leaving a smaller minority that felt that eval support was required. Richards (2010) did find that eval was used on real world websites but how much this applies to games outside of setting up a web specific environment is unclear. The worst case scenario would be that the compiler runtime would include a separate runtime that could be used to interpret code produced by eval at reduced speed, while requiring extra work and possibly negate numeric optimizations in many places it would remove most compatibility problems.

Developers were split on the issues surrounding undefined but seemed to welcome the idea of the compiler warning about potential undefined usages. Undefined behaviors creates problems for the type inference system yet the split opinions of developers warrant some extra investigation into deciding how to proceed on the issue. As undefined can cause propagated errors in JavaScript systems developers might have overestimated the impact of undefined but this cannot be verified until the compiler is more mature and used for real life projects. Compiler reporting on undefined usage should be investigated to see if the functionality would be more of an annoyance or aid to developers. The final options would be to either have full emulation of undefined behavior (with the performance penalty) or a limited emulation system based on magic values (numerics with a special NaN pattern and a special object for other cases).

The developers were also split on the issue of ahead of time fields, this issue is partially tied to the undefined issue so semi active emulation might be done either in the same way with the undefined magic numbers or by adding bitfields for the “existence” of particular fields. The bitfield updating method should provide most of the needed compatibility but put a small extra performance strain on field updates. This performance penalty would not be too bad since Richards(2010) showed a 6:1 ratio of field reads to writes and the cost of the bitfield updates would be comparable to that of garbage collection write barriers that are widely used in modern JIT runtimes.

The developers were not so worried about integer overflows but almost unanimously preferred compatibility in terms of generic execution and to either have the behavior enabled specifically or just do it manually by hand optimized Asm.JS style type coercions where needed. Implementing this according to developer wishes is what is done today and the

results are not mutually exclusive so the behavior in the current compiler is already acceptable.

Object literals was used by the majority with some small uptake of container types. With the widespread usage of this idiom it cannot be overlooked by a successful JavaScript compiler even if it complicates behavior, this was suspected beforehand and the compiler is already able to analyze the usage of object literals as a functionality to map keys to values but the exact runtime penalty of this has not yet been fully investigated.

The amount of developers who seemed to favor hand optimizations when it came to integers numbers was a bit unexpected but could be leveraged to attain even more performance or find alternative solutions to the trickier problems faced in increasing the compatibility of the code produced by the compiler. This could also be looked at in connection with the recent usage of Asm.JS to bring over useful C and C++ libraries for handling physics and other CPU heavy calculations in games. While the compiler is already capable of matching the optimizing JIT compilers in performance, it could actually gain a speed advantage since C and C++ libraries could be linked in natively using the same abstractions already used with Asm.JS without going through the Asm.JS translation steps that has a bit of a performance penalty at runtime.

8 Conclusion

While not complete the compiler subset so far implemented has been shown the potential to improve the performance situation compared to interpreters by even matching the performance of the contemporary optimizing JIT compilers. Fine tuning the system, unexplored optimizations and native code bindings directly linked into the binary shows that there is room to improve the performance figures further in real world scenarios, however the other hand questionnaire showed a need to revisit some compatibility assumptions that could potentially decrease performance. Overall the large performance improvement compared to interpreters speaks for themselves and implementing some of the compatibility fixes should not change the picture significantly compared to interpreter performance.

The biggest unresolved technical issue remaining to be investigated is to find out if there are common occurrences of pathological cases for the type inference system, this is since this hasn't been properly investigated while working on making the system capable of compiling the relatively small benchmarking cases.

With the performance goals met combined with the questionnaire responses, it seems that roughly half the JavaScript developers should find the compiler useful as designed today with roughly another quarter of the developers finding it suitable if the compatibility issues raised were sorted out.

In general these results combined shows enough promise to warrant future development of the system.

References

Agesen, O. (1995), The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. *In ECOOP '95, Ninth European Conference on Object-Oriented Programming, Århus, Denmark*

Agesen, O. (1996). Concrete type inference: delivering object-oriented applications, *Doctoral dissertation, Stanford University*

Agesen, O., Hölzle, U. (1995), Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages *OOPSLA '95 Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications* Pages 91-107

Appel, A. (1992), Compiling with continuations, *Cambridge University Press, ISBN 978-0-521-03311-4 (2006 paperback reprint)*

Cartwright, R., Fagen, M. (1991), Soft typing, *PLDI '91 Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* Pages 278 - 292

Cython (2014) Cython C-extensions for python
http://cython.org/ (Retrived 2014-12-11)

Dufour, M. , (2006), Shed Skin, An Optimizing Python-to-C++ Compiler , *Master thesis*
http://mark.dufour.googlepages.com/shedskin.pdf

Dufour, M , (2011), Shed Skin 0.9, blogpost.
http://shed-skin.blogspot.se/2011/09/shed-skin-09.html

Ecma International (2011) Standard ECMA-262, ECMAScript Language Specification, Edition 5.1 (compatible with ISO/IEC 16262:2011)
http://www.ecma-international.org/publications/standards/Ecma-262.htm (Retrived 2014-12-11)

Egorov, V (2013), Why asm.js bothers me
http://mrale.ph/blog/2013/03/28/why-asmjs-bothers-me.html (Retrived 2014-12-11)

Google, (2008) Chrome V8, Design Elements
https://developers.google.com/v8/design (Retrived 2014-12-11)

Hayen, K. (2014) Nuitka python compiler
http://nuitka.net/ (Retrived 2014-12-11)

Herman, D., Wagner, L. Zakai, A. (2013) Asm.JS specification working draft
http://asmjs.org/spec/latest/ (Retrived 2014-12-11)

LLVM Project (2014) Accurate Garbage Collection with LLVM
http://llvm.org/releases/3.6.0/docs/GarbageCollection.html (Retrived 2015-05-20)

Madsen, M., Sørensen, P., Kristensen, K., (2007), Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm, Master thesis
http://projekter.aau.dk/projekter/en/studentthesis/ecstatic--type-inference-for-ruby-using-thecartesian-product-algorithm%28e78517a5-e4cf-42d0-9caa-a80749e84c00%29.html (Retrived 2014-12-11)

Might, M. (2014) k-CFA: Determining types and/or control-flow in languages like Python, Java and Scheme
http://matt.might.net/articles/implementation-of-kcfa-and-0cfa/ (Retrived 2014-12-11)

Might, M. Shivers, O. (2006) Improving flow analyses via ΓCFA: Abstract garbage collection and counting. *11th ACM International Conference on Functional Programming (ICFP 2006). Portland, Oregon. September, 2006.* Pages 13--25.

Plevyak, J. (1988) OPTIMIZATION OF OBJECT-ORIENTED AND CONCURRENT PROGRAMS , *PHD dissertation,* University of Illinois at Urbana-Champaign, 1996

Reynolds, J. (1993), The discoveries of continuations, *LISP and Symbolic Computation, 1993, Vol.6(3), pp.233-247*

Salib, M. , (2004), Starkiller: A Static Type Inferencer and Compiler for Python , Master thesis
http://dspace.mit.edu/handle/1721.1/16688 (Retrived 2014-12-11)

Shivers, O., (1988), Control flow analysis in scheme , *PLDI '88 Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, Pages 164-174

Shivers, O. (1991) Control-Flow Analysis of Higher-Order Languages or Taming Lambda , *PHD dissertation* CMU-CS-91-145

Smaragdakis, Y., Bravenboer M., and Lhotak, O. (2011) Pick Your Contexts Well: Understanding Object-Sensitivity *ACM SIGPLAN Notices, 2011, Vol.46(1), pp.17*

Steele, G. (1978), RABBIT: A compiler for SCHEME, *Master Thesis, Massachusetts Institute of Technology*

Ratanaworabhan, P., Livshits, B., & Zorn, B. G. (2010). JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. *Proceedings of the 2010 USENIX conference on Web application development (pp. 3-3).*

Richards, G., Lebresne, S., Burg, B., & Vitek, J. (2010). An analysis of the dynamic behavior of JavaScript programs. *ACM Sigplan Notices (Vol. 45, No. 6, pp. 1-12).*

Van Emden, M. (2014) How recursion got into programming: a comedy of errors http://vanemden.wordpress.com/2014/06/18/how-recursion-got-into-programming-a-comedy-of-errors-3/ (Retrived 2014-12-11)

Vardoulakis, D., Shivers, O., (2011) CFA2: a Context-Free Approach to Control-Flow Analysis
http://arxiv.org/abs/1102.3676?frbrVersion=2 (Retrived 2014-12-11)

Wingolog, A., (2011) Value representation in JavaScript implementations
http://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations (Retrived 2014-12-11)

Appendix A: Glossary

HTML5	The latest HTML document standard for the web that also codifies approaches to incrementally add new functionality to browsers beside the core document standard.
API	Application Programming Interface, a standardized way of interacting with a resource.
Canvas	A simple 2D drawing API
WebGL	An adaptation of the OpenGL ES 2 standard for JavaScript used to render 2d and 3d graphics with the help of graphics chipset
FullScreen API	An API used to provide a way for elements in a web page to be expanded to the entire screen, suitable for movie viewing and games.
Native code	Computer code that is executed directly by the machine silicon without any other software in between to provide a translation.
Compilation	A description of the processes of turning the developers sources into code runnable by a computer. Most often done by the developer prior to distribution and execution and generally produces native code.
Interpretation	A usually slower method that runs the code in a form not native to the current machine by the native machine interpreting the non-native instruction stream.
Just in time compilation (JIT)	A generic description of modern and faster interpretation systems that speeds up execution by translating the code to be run into native code. In contrast with a regular compiler these kinds of systems defers the translation until the code needs to run. Benefits include faster testing and the ability of compilers to make decisions about how to generate code based on the actual data that needs to be processed.
Ahead of time compilation (AOT)	An explicit moniker of the classic compilation model of developers compiling code before distribution to end users.

Garbage Collection	Automatic reclaiming of memory and other resources when all references to a particular resource has disappeared.
Code signing	Applying a verifiable cryptographic signature to a piece of code to be run. By signing code the signer indicates to man and/or machine that this code has been vetted by the signing party and is assumed to be free of problems.
Typing	The difference for a computer system between a boolean (true or false), an integer(-1, 0, 1 ,2…), a floating point number (0.5, 3.141592), a string (“hello world”) and other kinds of object types.
Dynamic typing	Describing that code can have different kinds of data in the same position at different points in time.
Static typing	Contrary to the above this describes that there can only have one type in each position.
Explicit typing	Almost exclusively used with static typing, this is used to signify that the developer writes out the type to be used. f.ex: int A = 1+2; String B=”Hello”;
Implicit typing	The opposite of explicit typing but used with both dynamic and statically typed languages: var A=1+2; var B=”Hello”;
Soft typing	A kind of runtime system that outwardly to the developer works as a dynamically typed system but internally tries to constrain as much of the functionality as possible with static types.
Structural typing	A system where descriptions of what fields an object has rather than an identity is the important part, compare to Nominal typing
Nominal typing	A way of treating types where the identity of the type is the way to test equivalence,
Type inference	A process where a compiler infers static type information if possible from the source code.
Continuation	A continuation is a function defined as the rest of the program to be executed (Reynolds 1993)

Monomorphism	Cases where only single types are used in the same position
Polymorphism	Cases where multiple types are used in the same position

Appendix B: Benchmarking Samples

The code samples in this chapter should be directly runnable with node.js.

To run the code samples with default shells for Spidermonkey(JS), JavaScriptCore(jsc), Nashorn, Rhino and Duktape a small console shim is needed, the simplest one is the following:

```
console={log:print}
```

001_fib.js

```
var fib=function(x) {
	if (x<2)
		return x;
	else
		return fib(x-2)+fib(x-1);
};
console.log("warmup");
var i=0.0;
while(i<5) {
	fib(33);
	i=i+1;
}

start=new Date().valueOf();
var res=fib(35);
stop=new Date().valueOf();
console.log("fib result ",0|res);
console.log("Time:",stop-start);
```

002_fibco.js

```
var fib=function(x) {
	if (x<2)
		return x;
	else
		return (fib((x-2)|0)+fib((x-1)|0))|0;
};
console.log("warmup");
var i=0.0;
while(i<5) {
	fib(33);
	i=i+1;
}

start=new Date().valueOf();
var res=fib(35);
stop=new Date().valueOf();
console.log("fib result ",0|res);
console.log("Time:",stop-start);
```

003_cplx.js

```
var iter=100000000;

var cplx=function(i,mx,my) {
	var t=0;
	var ax=1.0;
	var ay=0.0;
	while(0<i) {
		t=ax*mx-ay*my;
		ay=ax*my+ay*mx;
		ax=t;
		i=(i-1)|0;
	}
	return {x:ax,y:ay};
};
console.log("warmup");
var i=0.0;
while(i<5) {
	cplx(iter,0.9,0.4358898943540674);
	i=i+1;
}

start=new Date().valueOf();
var res=cplx(iter,0.9,0.4358898943540674);
stop=new Date().valueOf();
console.log("Time:",stop-start);
console.log("cplx result ",res.x,res.y);
```

004_cplxo.js

```
var iter=50000000;

var cplx=function(i,m) {
	var t=0;
	var a={x:1,y:0};
	while(0<i) {
		t=a.x*m.x-a.y*m.y;
		a.y=a.x*m.y+a.y*m.x;
		a.x=t;
		i=(i-1)|0;
	}
	return a;
};
console.log("warmup");
var i=0.0;
while(i<5) {
	cplx(iter,{x:0.9,y:0.4358898943540674});
	i=i+1;
}

start=new Date().valueOf();
var res=cplx(iter,{x:0.9,y:0.4358898943540674});
stop=new Date().valueOf();
console.log("Time:",stop-start);
console.log("cplx result ",res.x,res.y);
```

005_cplxpo.js

```
var iter=10000000;

var cplx=function(i,a,m) {
	var t=0;
	while(0<i) {
		t=a.x*m.x-a.y*m.y;
```

```
                a.y=a.x*m.y+a.y*m.x;
                a.x=t;
                i=(i-1)|0;
        }
        return a;
};
var pcplx=function(iter,m) {
        var obj=[];
        obj.push({x:1,y:0});
        obj.push({y:0,x:1});
        obj.push({z:3,y:0,x:1});
        var i=0;
        var out=obj[0];
        var cf=cplx;
        while (i<obj.length) {
                out=cf(iter,obj[i],m);
                i++;
        }
        return out;
};
console.log("warmup");
var i=0.0;
while(i<5) {
        pcplx(iter,{x:0.9,y:0.4358898943540674});
        i=i+1;
}

start=new Date().valueOf();
var res=pcplx(iter,{x:0.9,y:0.4358898943540674});
stop=new Date().valueOf();
console.log("Time:",stop-start);
console.log("cplx result ",res.x,res.y);
```

006_array.js

```
var i=0.0;
var j=0;

var a=[];

//var max=10000000;
  var max=1000;
var iter=200000.0;
//var max=2;

while(i<max) {
        a.push(j);
        j=j^(j*374);
        j=(j+14513)&0xffff;
        i=i+1;
}

var test=function() {
        var i=0;
        var sum=0;
        while(i<max) {
                sum=sum+a[i];
                i=i+1;
        }
        return sum;
};

console.log("warmup");
i=0.0;
while(i<15) {
```

```
	test();
	i=i+1;
}

start=new Date().valueOf();
var res;
while(0<iter)
{
	res=test();
	iter=iter-1;
}
stop=new Date().valueOf();
console.log("test result ",0|res);
console.log("Time:",stop-start);
```

007_arrayco.js

```
var i=0.0;
var j=0;

var a=[];

//var max=10000000;
var max=1000;
var iter=200000.0;
//var max=2;

while(i<max) {
	a.push(j);
	j=j^(j*374);
	j=(j+14513)&0xffff;
	i=i+1;
}

var test=function() {
	var i=0;
	var sum=0;
	while(i<max) {
		sum=sum+a[i];
		i=(i+1)|0;
	}
	return sum;
};

console.log("warmup");
i=0.0;
while(i<15) {
	test();
	i=i+1;
}

start=new Date().valueOf();
var res;
while(0<iter)
{
	res=test();
	iter=iter-1;
}
stop=new Date().valueOf();
console.log("test result ",0|res);
console.log("Time:",stop-start);
```

Appendix C: Benchmarking results

The numbers here is the raw CSV data used to create the graphs. Each number corresponds to one sample of running time for calculating each benchmark in milliseconds

Win32

```
,,Test0,Test1,Test2,Test3,Test4
001_fib.js,ccref,93,109,109,109,93
001_fib.js,cm_in,266,265,265,265,265
001_fib.js,cm_no,312,312,327,327,312
001_fib.js,cmref,514,514,530,514,514
001_fib.js,v8no_c,218,218,218,218,219
001_fib.js,v8no_b,452,452,468,468,468
001_fib.js,sp_fa,110,109,111,110,110
001_fib.js,sp_ba,1233,1233,1240,1238,1231
001_fib.js,sp_in,14376,14400,14997,14270,14894
001_fib.js,nasho,578,436,578,421,468
001_fib.js,rhino,1482,1513,1560,1498,1544
001_fib.js,dukt,22308,21825,21886,23634,26193
002_fibco.js,ccref,78,93,78,78,93
002_fibco.js,cm_in,265,249,250,265,265
002_fibco.js,cm_no,328,328,327,343,343
002_fibco.js,v8no_c,218,218,218,218,218
002_fibco.js,v8no_b,562,578,578,578,578
002_fibco.js,sp_fa,118,118,119,118,118
002_fibco.js,sp_ba,1396,1395,1397,1393,1398
002_fibco.js,sp_in,16404,16409,16688,16702,17088
002_fibco.js,nasho,827,842,826,858,827
002_fibco.js,rhino,2199,2169,2184,2200,2262
002_fibco.js,dukt,28735,27721,28798,28283,27253
003_cplx.js,ccref,468,468,452,452,468
003_cplx.js,cm_in,468,452,468,453,452
003_cplx.js,cm_no,452,453,468,468,452
003_cplx.js,cmref,2502,2496,2500,2501,2511
003_cplx.js,v8no_c,468,452,452,452,452
003_cplx.js,v8no_b,4851,4852,4836,4852,4836
003_cplx.js,sp_fa,481,481,480,479,480
003_cplx.js,sp_ba,4071,4071,4066,4073,4084
003_cplx.js,sp_in,84315,83296,83218,83614,84160
003_cplx.js,nasho,1498,1607,1513,1466,1435
003_cplx.js,rhino,5148,5085,5132,5054,5148
003_cplx.js,dukt,51277,55021,53321,52853,54273
004_cplxo.js,cm_in,343,343,343,359,358
004_cplxo.js,cm_no,1576,1560,1591,1592,1560
004_cplxo.js,v8no_c,374,359,374,359,374
004_cplxo.js,v8no_b,4695,4712,4712,4696,4695
004_cplxo.js,sp_fa,378,377,377,377,378
004_cplxo.js,sp_ba,4681,4673,4681,4681,4698
004_cplxo.js,sp_in,105721,106027,105820,105670,105600
004_cplxo.js,nasho,1981,1997,2013,1950,1950
004_cplxo.js,rhino,7847,7987,7426,7753,8237
004_cplxo.js,dukt,87828,88280,87485,87626,87363
005_cplxpo.js,cm_in,219,218,218,219,218
005_cplxpo.js,cm_no,1030,1186,1092,1186,1030
005_cplxpo.js,v8no_c,374,374,374,374,374
005_cplxpo.js,v8no_b,2995,2995,2995,3011,2995
005_cplxpo.js,sp_fa,489,489,489,490,489
005_cplxpo.js,sp_ba,3170,3163,3170,3170,3168
005_cplxpo.js,sp_in,62709,62112,62092,62032,62089
005_cplxpo.js,nasho,1404,1404,1326,1420,1435
005_cplxpo.js,rhino,5133,5117,5132,5133,5132
005_cplxpo.js,dukt,52478,52432,52354,52369,52651
```

```
006_array.js,cm_in,702,718,718,702,718
006_array.js,cm_no,5741,7317,5772,5772,6256
006_array.js,v8no_c,452,468,452,468,452
006_array.js,v8no_b,1919,1935,1950,1950,1935
006_array.js,sp_fa,277,278,280,278,277
006_array.js,sp_ba,5719,5715,5711,5715,5740
006_array.js,sp_in,105456,105702,108196,105453,109719
006_array.js,nasho,5506,5757,5569,5584,5555
006_array.js,rhino,8346,8564,8424,8502,8673
006_array.js,dukt,179729,155969,174938,158808,151851
007_arrayco.js,cm_in,296,281,281,281,281
007_arrayco.js,cm_no,2075,2074,2074,2075,2075
007_arrayco.js,v8no_c,452,468,453,468,453
007_arrayco.js,v8no_b,2356,2324,2371,2325,2340
007_arrayco.js,sp_fa,371,372,372,372,373
007_arrayco.js,sp_ba,5886,5900,5884,5884,5901
007_arrayco.js,sp_in,122590,120525,116760,117357,117066
007_arrayco.js,nasho,6302,6209,6489,6349,6272
007_arrayco.js,rhino,11170,11419,11404,11435,11341
007_arrayco.js,dukt,197496,187154,202520,186826,190429
```

Raspberry PI B1

```
,,Test0,Test1,Test2,Test3,Test4
001_fib.js,ccref,1510,1520,1520,1520,1520
001_fib.js,cm_in,2876,2899,2885,2871,2875
001_fib.js,cm_no,3784,3795,3786,3786,3813
001_fib.js,cmref,6540,6550,6410,6420,6520
001_fib.js,v8n6,6328,6334,6326,6332,6328
001_fib.js,v8n12,11174,11061,11035,11170,11522
001_fib.js,sp_fa,1919,1940,1938,1918,1918
002_fibco.js,ccref,550,560,550,550,560
002_fibco.js,cm_in,2001,1997,1998,1997,2037
002_fibco.js,cm_no,3079,3076,3079,3079,3079
002_fibco.js,v8n6,7694,7701,7695,7722,7720
002_fibco.js,v8n12,12920,13023,13238,13132,12822
002_fibco.js,sp_fa,1890,1886,1885,1888,1886
003_cplx.js,ccref,4070,4070,4070,4070,4070
003_cplx.js,cm_in,2907,2910,2935,2907,2907
003_cplx.js,cm_no,2914,2914,2907,2911,2910
003_cplx.js,cmref,51920,48770,50450,50430,50920
003_cplx.js,v8n6,109432,109403,109460,109497,109689
003_cplx.js,v8n12,103421,103506,103334,103142,103266
003_cplx.js,sp_fa,5676,5672,5676,5700,5672
004_cplxo.js,cm_in,1600,1600,1600,1598,1600
004_cplxo.js,cm_no,20469,20563,20472,20485,20465
004_cplxo.js,v8n6,96322,96459,96264,96178,97580
004_cplxo.js,v8n12,122829,122868,123093,122843,122891
004_cplxo.js,sp_fa,6127,6143,6142,6126,6116
005_cplxpo.js,cm_in,969,960,963,963,961
005_cplxpo.js,cm_no,12375,12361,12381,12389,12355
005_cplxpo.js,v8n6,86042,82960,86269,85661,92117
005_cplxpo.js,v8n12,76343,76407,76407,76496,76310
005_cplxpo.js,sp_fa,5277,5274,5272,5281,5275
006_array.js,cm_in,16806,16671,16655,16653,16646
006_array.js,cm_no,42082,42069,42099,42077,42092
006_array.js,v8n6,30373,30378,30409,30498,30373
006_array.js,v8n12,31498,31435,31450,31438,31413
006_array.js,sp_fa,5575,5598,5570,5570,5599
007_arrayco.js,cm_in,6142,6140,6164,6178,6146
007_arrayco.js,cm_no,30131,30131,30164,30244,30132
007_arrayco.js,v8n6,32738,32758,32756,32756,32768
007_arrayco.js,v8n12,33767,33776,33609,33667,33595
007_arrayco.js,sp_fa,6223,6196,6155,6157,6181
001_fib.js,sp_ba,19206,19486
```

```
002_fibco.js,sp_ba,21315,21277
003_cplx.js,sp_ba,55214,55214
004_cplxo.js,sp_ba,70780,70681
005_cplxpo.js,sp_ba,49949,50003
006_array.js,sp_ba,103444,103158
007_arrayco.js,sp_ba,125522,125756
001_fib.js,sp_ba,23289,23413
001_fib.js,sp_in,204498,186132
001_fib.js,dukt,1131094,1064052
002_fibco.js,sp_ba,25690,26517
002_fibco.js,sp_in,221071,194745
002_fibco.js,dukt,1391052,1410797
003_cplx.js,sp_ba,66883,66645
003_cplx.js,sp_in,376200,375370
003_cplx.js,dukt,1428458,1504127
004_cplxo.js,sp_ba,85316,85619
004_cplxo.js,sp_in,1186910,1388429
004_cplxo.js,dukt,3306682,3292335
005_cplxpo.js,sp_ba,60601,60259
005_cplxpo.js,sp_in,801553,856546
005_cplxpo.js,dukt,2012607,1957465
006_array.js,sp_ba,125611,125497
006_array.js,sp_in,1704134,1668996
006_array.js,dukt,5892989,6649120
007_arrayco.js,sp_ba,195697,152099
007_arrayco.js,sp_in,2074499,1703201
007_arrayco.js,dukt,8058768,7355199
```

Raspberry PI B2

```
,,Test0,Test1,Test2,Test3,Test4
001_fib.js,ccref,760,760,760,760,760
001_fib.js,cm_in,2091,2090,2090,2091,2090
001_fib.js,cm_no,2779,2778,2775,2776,2777
001_fib.js,cmref,4250,4250,4250,4250,4250
001_fib.js,v8no_c,2456,2458,2456,2455,2457
001_fib.js,v8no_b,6795,6835,6793,6794,6793
001_fib.js,sp_fa,1629,1628,1628,1629,1628
001_fib.js,sp_ba,20759,20873,20759,20754,20754
001_fib.js,jsc_d,1997,1997,1996,1996,1997
001_fib.js,jsc_j,12408,12720,12367,12411,12354
001_fib.js,jsc_i,26957,26956,26965,26954,26944
001_fib.js,nasho,11794,11386,11477,11335,11480
002_fibco.js,ccref,490,490,490,490,490
002_fibco.js,cm_in,1733,1734,1734,1734,1732
002_fibco.js,cm_no,2395,2392,2393,2392,2396
002_fibco.js,v8no_c,2454,2454,2457,2457,2456
002_fibco.js,v8no_b,8516,8511,8512,8512,8511
002_fibco.js,sp_fa,1536,1537,1537,1536,1537
002_fibco.js,sp_ba,22797,22903,22900,22899,22901
002_fibco.js,jsc_d,2042,2041,2040,2044,2041
002_fibco.js,jsc_j,13373,13374,13376,13371,13374
002_fibco.js,jsc_i,30794,30807,30795,31157,30798
002_fibco.js,nasho,15626,15730,15877,15731,15634
003_cplx.js,ccref,4610,4610,4610,4610,4610
003_cplx.js,cm_in,3929,3929,3929,3929,3932
003_cplx.js,cm_no,3928,3929,3929,3928,3929
003_cplx.js,cmref,24350,24350,24350,24350,24350
003_cplx.js,v8no_c,4613,4614,4612,4613,4613
003_cplx.js,v8no_b,64524,64064,64116,64132,64151
003_cplx.js,sp_fa,5640,5637,5637,5637,5639
003_cplx.js,sp_ba,65646,65285,65294,65289,65304
003_cplx.js,jsc_d,9056,9057,9059,9058,9057
003_cplx.js,jsc_j,24478,24100,24459,24114,24095
003_cplx.js,jsc_i,82918,82957,82908,82929,82920
```

```
003_cplx.js,nasho,50807,51190,50836,50825,50834
004_cplxo.js,cm_in,2089,2050,2052,2050,2050
004_cplxo.js,cm_no,19820,19822,19823,19824,19822
004_cplxo.js,v8no_c,4273,4272,4272,4270,4272
004_cplxo.js,v8no_b,81846,81832,81780,81858,81817
004_cplxo.js,sp_fa,4449,4446,4447,4450,4447
004_cplxo.js,sp_ba,86228,86131,86120,86131,85776
004_cplxo.js,jsc_d,5726,5726,5727,5727,5726
004_cplxo.js,jsc_j,33846,33898,33905,33902,33846
004_cplxo.js,jsc_i,81452,81444,81494,81815,81798
004_cplxo.js,nasho,62222,62219,61108,62196,61556
005_cplxpo.js,cm_in,1229,1230,1229,1231,1229
005_cplxpo.js,cm_no,11994,11994,11992,11993,11992
005_cplxpo.js,v8no_c,6219,6220,6219,6220,6220
005_cplxpo.js,v8no_b,50169,50198,50592,50186,50200
005_cplxpo.js,sp_fa,4723,4726,4728,4727,4726
005_cplxpo.js,sp_ba,57909,57505,57486,57862,57495
005_cplxpo.js,jsc_d,13452,13454,13450,13453,13456
005_cplxpo.js,jsc_j,25027,25100,24939,24931,25052
005_cplxpo.js,jsc_i,48868,49283,49237,48874,48878
005_cplxpo.js,nasho,41732,41752,41774,41718,42110
006_array.js,cm_in,7883,7881,7881,7886,7884
006_array.js,cm_no,24053,24015,24005,24010,24010
006_array.js,v8no_c,4469,4469,4468,4468,4468
006_array.js,v8no_b,22247,22243,22248,22254,22621
006_array.js,sp_fa,4810,4810,4813,4811,4813
006_array.js,sp_ba,110679,102806,103180,102819,103218
006_array.js,jsc_d,8250,8252,8247,8247,8246
006_array.js,jsc_j,42230,42226,42228,42227,42599
006_array.js,jsc_i,140438,140791,140471,140874,140456
006_array.js,nasho,144238,143528,144542,143920,143794
007_arrayco.js,cm_in,4802,4805,4802,4805,4804
007_arrayco.js,cm_no,18877,18880,18872,19247,18876
007_arrayco.js,v8no_c,4469,4468,4470,4469,4471
007_arrayco.js,v8no_b,24649,24641,24654,24654,24639
007_arrayco.js,sp_fa,5492,5491,5490,5493,5491
007_arrayco.js,sp_ba,114751,118252,114809,118312,114750
007_arrayco.js,jsc_d,8250,8247,8247,8248,8247
007_arrayco.js,jsc_j,44622,44611,44621,44993,44620
007_arrayco.js,jsc_i,161469,161829,161749,161383,164269
007_arrayco.js,nasho,170721,170952,171143,170867,171380
001_fib.js,sp_in,78937,79057
001_fib.js,rhino,26269,26253
001_fib.js,dukt,495224,461889
002_fibco.js,sp_in,84915,85141
002_fibco.js,rhino,35472,35778
002_fibco.js,dukt,626624,643804
003_cplx.js,sp_in,267937,267954
003_cplx.js,rhino,136147,135926
003_cplx.js,dukt,954121,953236
004_cplxo.js,sp_in,560811,559066
004_cplxo.js,rhino,170031,169823
004_cplxo.js,dukt,1818710,1831224
005_cplxpo.js,sp_in,338900,338950
005_cplxpo.js,rhino,114481,115004
005_cplxpo.js,dukt,1094543,1094539
006_array.js,sp_in,538550,520576
006_array.js,rhino,236915,237215
006_array.js,dukt,3377678,3773322
007_arrayco.js,sp_in,547954,535538
007_arrayco.js,rhino,299922,297781
007_arrayco.js,dukt,3816090,3823619
```

C compilers

```
,,Test0,Test1,Test2,Test3,Test4
001_fib.js,msvc_05,240,240,230,250,240
001_fib.js,msvc_13,250,260,260,260,260
001_fib.js,clang,343,345,343
001_fib.js,gcc,190,190,190,190,190
002_fibco.js,msvc_05,240,270,240,240,240
002_fibco.js,msvc_13,270,260,260,272,260
002_fibco.js,clang,260,250,260,250,250
002_fibco.js,gcc,200,200,200,200,210
003_cplx.js,msvc_05,460,460,460,470,460
003_cplx.js,msvc_13,460,470,460,470,460
003_cplx.js,clang,511,510,520,510,520
003_cplx.js,gcc,511,520,508,511,500
004_cplxo.js,msvc_05,390,390,378,380,381
004_cplxo.js,msvc_13,360,355,360,352,360
004_cplxo.js,clang,250,260,260,260,250
004_cplxo.js,gcc,240,230,230,230,230
005_cplxpo.js,msvc_05,230,230,230,230,230
005_cplxpo.js,msvc_13,220,224,220,220,220
005_cplxpo.js,clang,220,220,220,220,220
005_cplxpo.js,gcc,140,140,139,138,140
006_array.js,msvc_05,930,920,930,928,940
006_array.js,msvc_13,715,725,720,722,720
006_array.js,clang,880,880,880,885,880
006_array.js,gcc,1770,1758,1762,1760,1770
007_arrayco.js,msvc_05,620,610,610,610,600
007_arrayco.js,msvc_13,290,290,290,290,288
007_arrayco.js,clang,370,370,370,370,370
007_arrayco.js,gcc,280,280,280,280,275
```

Appendix D: Questionnaire

JavaScript static compilation questions
*Required

Firstly, do you program in plain JavaScript or use some kind of transpiler? *

- ○ JavaScript
- ○ CoffeeScript
- ○ TypeScript
- ○ Haxe
- ○ Other: ☐

What JavaScript/EcmaScript environments have you used? *

- ☐ Chrome/Node (V8)
- ☐ Firefox (Spidemonkey,etc)
- ☐ Safari (JavaScriptcore)
- ☐ Internet explorer
- ☐ Rhino (JVM)
- ☐ Nashorn (JVM)
- ☐ PhoneGap / Cordova
- ☐ Titanium Appcelerator
- ☐ CocoonJS
- ☐ GameClosure

- ☐ Other:

Dynamic code loading

Does your code use eval? *

- ○ Yes
- ○ No

Does your code use another mechanism for code loading in a dynamic fashion? *

Such as using the CommonJS function require("level"+number) ? or the rhino load("level"+number)

- ○ Yes
- ○ No

If you answered yes to the eval question above, could your code easily be changed to use load() or CommonJS require() as described above?

- ○ Yes
- ○ No

If you use dynamic code loading and like to add anything to the above, do so here.

2. Undefined behaviour

Does your code depend on undefined for it's functionality? *

(Excluding explicit "undefined" testing for browser functionality to retain compability)

- ○ Yes
- ○ No

Would your code behave differently if a variable only used as a number upon initialization would be set to 0 or NaN instead of undefined ? *

- ○ Yes
- ○ No

Would your code behave differently if a variable only used for objects would be null instead of undefined? *

- ○ Yes
- ○ No

Would you consider it a benefit if the compiler faulted on most usages of undefined?

For example any numeric operation with undefined would produce an error at compiletime instead of a NaN at runtime.

- ○ Yes
- ○ No
- ○ Other:

If you have additional thoughts about undefined usage or behaviour write here

3. Object behaviour

Would your code behave differently if fields existed ahead of time? *

Ie, var obj={}; var beforeDefinition=obj.hasOwnProperty("a"); obj.a=true; console.log(beforeDefinition); // with beforeDefinition being true instead of false as with regular Ecmascript/Javascript

- ○ Yes
- ○ No

Numeric behaviour

The ecmascript standard specifies that all numbers are handled as IEEE-754 double precision floating point numbers, however processors usually handle integers faster than double precision numbers and in many cases programmers only expect the integer behaviour in most cases when using numbers. JavaScript JIT compilers can easily handle overflowing integers by promoting to doubles at runtime and recompiling code but an ahead of time compiler has no such luxury. A mode similar to how C uses numbers with separate int and double types and promotion rules similar to C is looked at.

Would your code be affected by integer overflows? *

Answer conservatively if you are unsure. For example (1<<30)+(1<<30) becomes -1 for 32 bit integers and 65536*65536 would produce 0 for a 32bit integer.

- ○ Yes
- ○ No

To get the benefits of integer operations what would be your preference? *

- ○ Have integer operations separated from doubles by default (fast, less compatibility, risk of subtle bugs not found on other implementations)
- ○ Have doubles(compatible) as the default but enable a similar fast mode for specified functions and other blocks (similar to the "use strict" and "use asm" directives)
- ○ Have hand optimized code sequences (no compiler option, programmer specifices coercions manually by for example 0|(a+b), similar to how asm.js requires operations to be written)
- ○ Other:

Code style

To store dynamic mappings between keys and values, do you use plain object literals or some kind of container class functionality? *

Ie. create an object var namedColors={}; and then DYNAMICALLY load in values in the fashion of namedColors[colorID] =colorNumber; where in one instance might be colorID="red" and colorNumber=0xff0000; to create a mapping or does your code look more like var namedColors=new HashMap(); namedColors.put(colorID,colorNumber);

- ○ Object literals ({})
- ○ Container classes (new HashMap, new TreeMap, etc)
- ○ Other: