JavaScript AOT Compilation

Manuel Serrano
Inria/Université Côte d'Azur
Manuel.Serrano@inria.fr

Abstract

Static compilation, *a.k.a.,* ahead-of-time (AOT) compilation, is an alternative approach to JIT compilation that can combine good speed and lightweight memory footprint, and that can accommodate read-only memory constraints that are imposed by some devices and some operating systems. Unfortunately the highly dynamic nature of JavaScript makes it hard to compile statically and all existing AOT compilers have either gave up on good performance or full language support. We have designed and implemented an AOT compiler that aims at satisfying both. It supports full unrestricted ECMAScript 5.1 plus many ECMAScript 2017 features and the majority of benchmarks are within 50% of the performance of one of the fastest JIT compilers.

ACM Reference format:
Manuel Serrano. 2018. JavaScript AOT Compilation. In *Proceedings of ACM SIGPLAN Dynamic Language Symposium, Boston, NY, USA, November 6, 2018 (DLS'18),* 14 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Nowadays, JavaScript is no longer confined to the programming of web pages. It is also used for programming server-side parts of web applications, compilers (Microsoft 2013), and there is a growing trend for using it for programming internet-of-things (IoT) applications. All major industrial actors of the field are looking for, or are already providing, JavaScript based development kits (IoT.js, Espruino, JerryScript, Kinoma.js, ...). In this application domain, JavaScript programs execute on small devices that have limited hardware capacities, for instance only a few kilobytes of memory. Just-in-time (JIT) compilation, which has proved to be so effective for improving JavaScript performances (Chang et al. 2009; Chevalier-Boisvert and Feeley 2015, 2016; Gal et al. 2009), is unthinkable in these constrained environments. There would be just not enough memory nor CPU capacity to execute them at runtime. Furthermore memory write operations on executable segments are sometimes impossible on the devices, either because of the type of memory used (ROM or FLASH) or simply because the operating system forbids them (iOS for instance). Pure JavaScript interpreters are then used, but this comes with a strong performance penalty, especially when compared to assembly or C programs, that limits the possible uses.

DLS'18, Boston, NY, USA
2018. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

When JIT compilation is not an option and when interpretation is too slow, the alternative is static compilation, also known as ahead-of-time (AOT) compilation. However, this implementation technique seems not to fit the JavaScript design whose unique combination of antagonistic features such as functional programming support, high mutation rates of applications, introspection, and dynamicity, makes most known classical AOT compilation techniques ineffective.

Indeed, JavaScript is hard to compile, much harder than languages like C, Java, and even harder than Scheme and ML two other close functional languages. This is because a JavaScript source code accepts many more possible interpretations than other languages do (Gong et al. 2014). It forces JavaScript compilers to adopt a defensive position by generating target codes that can cope with all the possible, even unlikely, interpretations.

Let us illustrate this problem with a running example. It shows well-known JavaScript folklore, but readers unfamiliar with this language might find it helpful to grasp the challenges that must be overcome to implement JavaScript efficiently. Let us consider the following statement:

```
while( i < a.length ) sum += a[ i++ ];
```

It seems innocuous, especially to C and Java programmers that will see here nothing more than a mundane loop summing all the elements of an array. The assembly code that compilers are expected to generate, is easy to guess as a mere isomorphic translation would already be efficient.

With JavaScript, this is another story. Of course, as with C and Java, the code fragment can be used to sum array elements, but it can also be used differently, for instance for summing the attributes of an arbitrary object as in:

```
var a = { length: 3; "1": 1, "3": 3, "2": 2 };
while( i < a.length ) sum += a[ i++ ];
```

This is worrisome for the compiler that cannot assume that the elements to sum are stored consecutively in memory. To keep the thing relatively simple, let us assume that `a` is indeed an array. The problem posed to the compiler is still far from simple. JavaScript supports *sparse arrays*, so `a` being an array is not enough for assuming that the elements are aligned. Even worse, because of the prototype chaining, it might well be that some elements are stored in another object, as in:

```
Array.prototype[ "2" ] = 28;
var a = [ 1,,3 ];
delete a[ 2 ];
while( i < a.length ) sum += a[ i++ ];
```

Since the element `2` is removed from `a`, the prototype chain is followed when the element `a[2]` is fetched, which yield the value `28`. Obviously, with such data structures, the assembly code generated for this JavaScript loop is unlikely to look like a mere iteration over a couple of assembly instructions that fetch and sum consecutive memory locations!

The index `i` also deserves some attention as nothing imposed it to be an integer. For instance, it could well be a floating-point number or, more interestingly, a string, or any other object that will then be converted into either a number or a string at runtime. For the example, let us consider that `i` is a string. In that case, the `i++` expression first converts the initial string into a number and then increments it.

Even more challenging, let us replace `i++` with `i+=1`. In JavaScript `i++`, `i+=1`, and `i=i+1` are not all semantically equivalent, when type conversions are involved. When `i` is a string, `i+=1` no longer adds the integer 1 but it appends a suffix "`1`" to the string. This enables a yet another input data type to be used with the program. For instance, the following program, also adds 3 (0+1+2) to the `sum` variable

```
var i = "0"
var a = { length: "011", "0": 0, "01": 1, "011": 2 };
while( i < a.length ) sum += a[ i += 1 ];
```

Notice that it is still possible and correct, to use this version of the loop, with `a` being an array and `i` being an integer! Finally, notice that we have only studied the modifications of the type and shape of variables `i` and `a`, but of course it might be possible as well to use various types for the variable `sum`, with other interesting challenges for the compiler.

This example illustrates that in general compilers can assume very little about JavaScript programs. The situation is worsened further by the *raise as little errors as possible* principle that drives the design of the language. JavaScript functions are not required to be called with the declared number of arguments, fetching an unbound property is permitted, assigning undeclared variables is possible, etc.

All these difficulties seem to prevent classic static compilers to deliver efficient code for a language as dynamic and as flexible as JavaScript. We do not share this point of view. We think that by carefully combining classical analyses, by developing new ones when needed, and by crafting a compiler where the results of the high-level analyses are propagated up to the code generation, it is possible for AOT compilation to be in the same range of performance as fast JIT compilers. This is what we attempt to demonstrate with this study. Of course, our ambition is not to produce a compiler strictly as fast as the fastest industrial JavaScript implementations. This would require much more engineering strength than what we can afford. Instead, we only aim at showing that static compilation can have performances reasonably close to those of fastest JavaScript implementations. *Reasonably close* is of course a subjective notion, that everyone is free to set for himself. For us, it means a compiler showing half the performances of the fastest implementations. We will show in Section 7 that this objective has been reached or is closed to be reached for many benchmarks.

We have developed such a static compiler. It is named Hopc. It supports the *full unrestricted* ECMAScript 5.1 and many features of ECMAScript 6 (ECMA International 2015). It passes all the ECMAScript 5.1 test262-51 (ECMA International 2016) test suite and many other compatibility tests such as the Kangax test suite (Kangax 2018), the MDN examples, and most Nodejs tests. This paper focuses on how it uses type information to generate efficient code. It is organized as follows. Section 2 presents the main structure of the compiler. Sections 3-6 overview the typing analyses we have developed and some parts of the code generator. Section 7 presents a performance evaluation and Section 8 presents the related work.

2 AOT Compilation

The syntax of a JavaScript program gives a general information about its meaning, but it gives almost no clue about the nature of the values it manipulates, as the `while` loop example of Section 1 that has been used with all sorts of data. JIT compilers wait until they receive the actual values to compile a program. When they generate the target code, they know both the static structure of the program and the dynamic memory layout of the data. So, they are able to approach the performance of compilers of more static languages that know these two things statically.

We have accommodated the principle of JIT compilation customization (Chambers and Ungar 1989) to AOT compilation by generating two versions for each function: a generic version that can cope with all the possible interpretations, and an optimized customized version, specialized for specific data representation. In order to decide which customized versions to generate, Hopc extracts as much as possible information from the source code. Modular compilation, *a.k.a.,* separate compilation, prevents it to always being able to make such deductions. In that case, it *speculates* beforehand on the data structures that are likely to be used by the exported functions. The key principle of the speculation is the following assumption. *The most likely data structure a program will use is the one for which the compiler is able to produce its best code.*

The intuition behind the speculation is that the best code will be generated for stable, simple, and classical data structures that are used for implementing classical algorithms. In other words, the data structures that are likely to correspond to the programmer's intentions. In the example of Section 1, many data types and many usages are possible. However, the most likely one is the iteration over a flat array of numbers. This is the principle that governs Hopc design.

Hopc uses many analyses and optimizations. Some are simple adaptations to JavaScript of existing analyses. These are not presented in this paper because already described in the literature. For instance, the *Scope Narrowing* optimization

that is a pre-requisite to all other analyses is not presented here because it is a direct adaptation of Scheme's `letrec` compilation (Ghuloum and Dybvig 2009; Waddell et al. 2005). Others are new or adapted to fit JavaScript.

- *Data-flow Type Analysis* performs a type analysis that is used to establish the hint typing hypotheses. It is based on the occurrence typing (Kent et al. 2016; Tobin-Hochstadt and Felleisen 2010) but it is presented here to show that in spite of the JavaScript highly dynamic nature, useful type information can be computed statically.
- *Hint Typing* is the speculative type inference. It selects the hypotheses under which the compiler is able to deliver its best code, and from these hypotheses, it deduces types that let the compiler decide which specialized versions to generate. Hint types are unsound as they do not denote super sets of all the possible values variables can hold at runtime, neither they abstract all possible correct program executions. When they do not, the main negative consequence is a waste of space caused by unused specialized code versions.
- *Integer Range Analysis* approximates statically integer operations from the result of the hint typing analysis. The main originality of this contribution is not the analysis itself that is based on (Logozzo and Venter 2010) but the way its results are used by the code generator to avoid boxing and tagging and to implement array indices efficiently.

Type information alone is not enough to generate fast code. For that, the compilation chain has to include all sorts of optimizations such as inline property caches, closure allocation, function inlining, data-flow analysis, register allocation, etc. Hopc includes many of them but it is out of the scope of the present paper to describe them all. We have chosen to only focus on the code generation for array accesses (Section 6) because it is a difficult and characteristic problem posed by the JavaScript compilation and also because arrays are ubiquitous in JavaScript programs.

3 Data-flow Type Analysis

Types improve the quality of the generated code by enabling the compiler to eliminate runtime tests and to use efficient data representations. In Hopc, types are collected using four different analyses, each refining the information collected by the others. The fist analysis is a *data-flow type analysis*. It is based on the occurrence typing (Kent et al. 2016; Tobin-Hochstadt and Felleisen 2010).

The *Data-Flow Type Analysis* distinguishes variable declarations and variable references. It assigns a potentially different type to each occurrence of the same variable. For that, it follows the execution flow and gathers the explicit *and* implicit type information the program contains. Let us illustrate its behavior on the following definition:

```
1 function toString( o ) {
2    if( typeof o == "string" ) return o;
3    if( typeof o == "number" ) return o + "";
4    if( "toString" in o ) return o.toString();
5    return toString( JSON.stringify( o ) );
6 }
```

The variable `o` is *explicitly* checked on line 2. In the positive branch the type `string` is associated to the variable. For a similar reason it is associated the type `number` on line 3. On line 4, it is only checked if `toString` is a property of `o`. This implies that in the following execution flow `o` is of type `object` (line 4 and 5), as the operator `in` requires its right-and-side argument to be an object, and raises an error otherwise. We call this an *implicit* type check.

3.1 Typing algorithm

For the sake of simplicity, we present a simplified version of the algorithm actually implemented in Hopc. We consider only a subset of the actual source language, where `throw`, `break`, and `continue` are omitted and where `return` is considered always in tail position. Under these assumptions, the typing algorithm can be presented as a set of inference rules. The language we consider is:

value	v	::=	**nil** | *bool* | …
expr	e	::=	v | x | $e(e)$ | $e[e]$ | **new** e | **function**(x) $\{s\}$
stmt	s	::=	x=e | s;s | **if**(e) $\{s\}$ **else** $\{s\}$
			| **while**(e) $\{s\}$ | **return** e
type	τ	::=	$\top$ | nil | bool | object | … | $\tau \rightarrow \tau$

Γ is a *type environment* mapping variables to types. Each program location is associated with one type environment. A type judgment is of the form $[\Gamma \vdash inst \implies \langle \tau, \Gamma' \rangle]$, meaning that the instruction *inst* is of type τ and evaluating that instruction modifies the type environment into Γ'. As statements are only evaluated for their side-effect, they produce no value and their judgments are written $[\Gamma \vdash s \implies \langle \bot, \Gamma \rangle]$. The core typing rules are given in Figure 1. All types are subtypes of the type `any` also noted $\top$. $\bot$ is a fake type designating the lack of precise type information. We denote $\Gamma_1 \sqcup \Gamma_2$ the least upper bound of Γ_1 and Γ_2. $\Gamma \backslash \{x\}$ designates the environment where the variable x is filtered out. When an expression e evaluates to a function, we denote $e_{\downarrow body}$ its body.

The typing algorithm does not keep track of values stored in objects. As such, the typing of the property access Property is typed with $\top$. Typing object properties has the potential of significantly helping the compiler producing better code. It will be studied in future work. Function calls are split in two rules. Call is used when the called function is known statically. The type of the expression is the return type of the function and the typing environment is extended with that of the called function. Funcall is used otherwise. The type of the expression is unknown ($\bot$) and all mutated variables are stripped of the typing environment. In our actual implementation, only the mutated global variables and the mutated local variables that appeared free in at least one

$\boxed{\Gamma \vdash e \Rightarrow \langle \tau, \Gamma' \rangle}$

$$\frac{}{\Gamma \vdash \texttt{nil} \Rightarrow \langle \texttt{nil}, \Gamma \rangle}\ (\textsc{Nil}) \qquad \frac{}{\Gamma \vdash \texttt{true} \Rightarrow \langle \texttt{bool}, \Gamma \rangle}\ (\textsc{True}) \qquad \frac{}{\Gamma \vdash \texttt{false} \Rightarrow \langle \texttt{bool}, \Gamma \rangle}\ (\textsc{False}) \qquad \frac{x \in Dom(\Gamma)}{\Gamma \vdash x \Rightarrow \langle \Gamma(x), \Gamma \rangle}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash e \Rightarrow \langle \tau, \Gamma_1 \rangle}{\Gamma \vdash x{=}e \Rightarrow \langle \tau, \Gamma_1[x:\tau] \rangle}\ (\textsc{Assign}) \qquad \frac{}{\Gamma \vdash \textbf{function}(x)\ \{s\} \Rightarrow \langle \bot \to \bot, \Gamma \rangle}\ (\textsc{Abs}) \qquad \frac{\Gamma \vdash e_2 \Rightarrow \langle \tau_2, \Gamma_2 \rangle \quad \Gamma_2 \vdash e_1 \Rightarrow \langle \tau_1, \Gamma_1 \rangle}{\Gamma \vdash e_1[e_2] \Rightarrow \langle \top, \Gamma_1 \rangle}\ (\textsc{Property})$$

$$\frac{\Gamma \vdash e \Rightarrow \langle \tau, \Gamma_1 \rangle}{\Gamma \vdash \textbf{return}\ e \Rightarrow \langle \bot, \Gamma \rangle}\ (\textsc{Return}) \qquad \frac{\Gamma \vdash e \Rightarrow \langle \tau, \Gamma_1 \rangle}{\Gamma \vdash \textbf{new}\ e \Rightarrow \langle \texttt{object}, \Gamma_1 \rangle}\ (\textsc{New}) \qquad \frac{\Gamma \vdash s_1 \Rightarrow \langle \tau_1, \Gamma_1 \rangle \quad \Gamma_1 \vdash s_2 \Rightarrow \langle \tau_2, \Gamma_2 \rangle}{\Gamma \vdash s_1; s_2 \Rightarrow \langle \bot, \Gamma_2 \rangle}\ (\textsc{Seq})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \langle \bot \to \bot, \Gamma_1 \rangle \quad \Gamma_1 \vdash e_2 \Rightarrow \langle \tau_2, \Gamma_2 \rangle \quad \Gamma[x:\tau_2] \vdash e_{1\downarrow body} \Rightarrow \langle \tau, \Gamma_3 \rangle}{\Gamma \vdash e_1(e_2) \Rightarrow \langle \tau, \Gamma_3 \rangle}\ (\textsc{Call}) \qquad \frac{\Gamma \sqcup \Gamma_2 \vdash e \Rightarrow \langle \tau, \Gamma_1 \rangle \quad \Gamma_1 \vdash s \Rightarrow \langle \bot, \Gamma_2 \rangle}{\Gamma \vdash \textbf{while}(e)\ \{s\} \Rightarrow \langle \bot, \Gamma_1 \sqcup \Gamma_2 \rangle}\ (\textsc{While})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \langle \tau_1, \Gamma_1 \rangle \quad \tau_1 \neq \bot \to \bot \quad \Gamma_1 \vdash e_2 \Rightarrow \langle \tau_2, \Gamma_2 \rangle}{\Gamma \vdash e_1(e_2) \Rightarrow \langle \top, \Gamma_2 \backslash \{assigVars\} \rangle}\ (\textsc{Funcall}) \qquad \frac{\Gamma \vdash e \Rightarrow \langle \tau, \Gamma_1 \rangle \quad \Gamma_1 \vdash s_1 \Rightarrow \langle \bot, \Gamma_2 \rangle \quad \Gamma_1 \vdash s_2 \Rightarrow \langle \bot, \Gamma_3 \rangle}{\Gamma \vdash \textbf{if}(e)\ \{s_1\}\ \textbf{else}\ \{s_2\} \Rightarrow \langle \bot, \Gamma_2 \sqcup \Gamma_3 \rangle}\ (\textsc{If})$$

Figure 1. Data-flow core typing rules.

function are removed from the typing environment. For both calls, the potential side effects of the function expression and the argument expression are propagated to the produced type environment. Not accounted by the Call rule, when a constant function is called, the types of the actual argument values are accumulated. When a function is used as a reference, that is not in the syntactic position of a function in a call, the type $\top$ is assigned to its argument.

The If rule types separately its two branches. The type environment it produces is the least upper bound of the type environments computed for the two branches. To compute it, when two types are not strictly equal, they are merged into the type $\top$. The peculiarity of the While rule comes from the handling of the side effects that might occur when evaluating the test and the body, as illustrated by the following examples:

```
1 x = 4; while( x < 10 ) { x = "20"; }
2 x = 4; while( x = "20", false ) { x = true }
```

When entering the first `while`, the variable `x` is known to be an integer, after one iteration, its type has changed to `string`. After executing the second `while` the variable `x` is a string, as the body is never evaluated.

3.2 Collecting types

We now consider an extension of the core language

$$\text{expr} \quad e \quad ::= \quad \ldots \mid \textbf{typeof}\ e \mid e\ \textbf{instanceof}\ e \mid e == e \mid e\ \textbf{in}\ e$$

that enables the typing algorithm to infer types from the program control-flow by using the additional rules given in Figure 2.

The rules Typeof, TypeofTrue, and TypeofFalse are used for typing conditional expressions whose tests compare a variable value to a type name. These rules use the additional function $TName$ that maps a type to its external type name (for instance, in JavaScript the `bool` type is named `boolean`). The rule Typeof types the *then* branch of the conditional statement with an environment where the variable `x` is known to be of type τ. It applies when no precise type information is known about the variable. When the type of `x` is known then either TypeofTrue or TypeofFalse applies and only the live branch is typed.

The `instanceof` typing shares similarities with `typeof`. In the *then* branch of a test, the variable is known to be an `object`. Additionally the rule also assigns the type `function` to its right-hand-side expression, as it is required to be a function. This is visible in the VInstOfV rule. A similar reasoning is used for the Call rule, where the variable x is known to be a function in the rest of the evaluation, and for the in rule where x is known to be an `object`.

The other rules (Binop, SLength, and SIndexOf) are straightforward. Although not presented here, the compiler uses many other similar rules for typing known library functions and other operators.

3.3 Control flow breaks

The occurrence typing has been introduced in the context of the Scheme programming language, which is an expression-based functional language. JavaScript is statement-based and this demands to adapt the occurrence typing to cope with the control flow operators the language provides. Let us consider the following example:

```
1 function f() {
2    function h( x ) {
3       if( x > 0 ) throw x;
4       errno = 3;
5    }
6    errno = undefined;
7    h( 10 );
8    ... errno ...
9 }
```

According to the rules given in Figure 1, the variable `errno` is assigned the type `undefined` line 6. Following the call line 7, it is unconditionally assigned the type `number` line 4, which is wrong because the line is executed only when `x` is negative or null. By consequence, it is incorrectly considered of type `number` line 8. Fixing that problem can be solved simply by merely considering that all instructions that follow potential

$\boxed{\Gamma \vdash e \Rightarrow \langle \tau, \Gamma' \rangle}$

$$\frac{\Gamma \vdash x \Rightarrow \langle \gamma, \Gamma \rangle \quad \gamma = \bot \vee \gamma = \top \quad \Gamma[x:\tau] \vdash s_1 \Rightarrow \langle \bot, \Gamma_2 \rangle \quad \Gamma \vdash s_2 \Rightarrow \langle \bot, \Gamma_3 \rangle}{\Gamma \vdash \texttt{if}(x == \texttt{typeof}\ TName(\tau))\ \{s_1\}\ \texttt{else}\ \{s_2\} \Rightarrow \langle \bot, \Gamma_2 \sqcup \Gamma_3 \rangle}\ (\textsc{Typeof})$$

$$\frac{\Gamma \vdash x \Rightarrow \langle \tau, \Gamma \rangle \quad \Gamma \vdash s_1 \Rightarrow \langle \bot, \Gamma_2 \rangle}{\Gamma \vdash \texttt{if}(x == \texttt{typeof}\ TName(\tau))\ \{s_1\}\ \texttt{else}\ \{s_2\} \Rightarrow \langle \bot, \Gamma_2 \rangle}\ (\textsc{TypeofTrue})$$

$$\frac{\Gamma \vdash x \Rightarrow \langle \gamma, \Gamma \rangle \quad \gamma \neq \tau \wedge \gamma \neq \bot \wedge \gamma \neq \top \quad \Gamma \vdash s_2 \Rightarrow \langle \bot, \Gamma_2 \rangle}{\Gamma \vdash \texttt{if}(x == \texttt{typeof}\ TName(\tau))\ \{s_1\}\ \texttt{else}\ \{s_2\} \Rightarrow \langle \bot, \Gamma_2 \rangle}\ (\textsc{TypeofFalse})$$

$$\frac{\Gamma \vdash x \Rightarrow \langle \tau_1, \Gamma \rangle \quad \Gamma \vdash e \Rightarrow \langle \tau_2, \Gamma_2 \rangle \quad \Gamma_2[x:\texttt{object}] \vdash s_1 \Rightarrow \langle \bot, \Gamma_3 \rangle \quad \Gamma_2 \vdash s_2 \Rightarrow \langle \bot, \Gamma_4 \rangle}{\Gamma \vdash \texttt{if}(x\ \texttt{instanceof}\ e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \Rightarrow \langle \bot, \Gamma_3 \sqcup \Gamma_4 \rangle}\ (\textsc{vInstOf})$$

$$\frac{\Gamma \vdash x_1 \Rightarrow \langle \tau_1, \Gamma \rangle \quad \Gamma \vdash x_2 \Rightarrow \langle \tau_2, \Gamma_2 \rangle \quad \Gamma_2[x_1:\texttt{object}, x_2:\bot \to \bot] \vdash s_1 \Rightarrow \langle \bot, \Gamma_3 \rangle \quad \Gamma_2[x_2:\bot \to \bot] \vdash s_2 \Rightarrow \langle \bot, \Gamma_4 \rangle}{\Gamma \vdash \texttt{if}(x_1\ \texttt{instanceof}\ x_2)\ \{s_1\}\ \texttt{else}\ \{s_2\} \Rightarrow \langle \bot, \Gamma_3 \sqcup \Gamma_4 \rangle}\ (\textsc{vInstOfV})$$

$$\frac{\Gamma \vdash x \Rightarrow \langle \bot \to \tau, \Gamma \rangle \quad \Gamma \vdash e_2 \Rightarrow \langle \top, \Gamma_2 \rangle}{\Gamma \vdash x(e_2) \Rightarrow \langle \tau, \Gamma_2[x:\bot \to \tau] \rangle}\ (\textsc{Call}) \qquad \frac{\Gamma \vdash e_1 \Rightarrow \langle \texttt{string}, \Gamma_1 \rangle \quad \Gamma_1 \vdash e_2 \Rightarrow \langle \tau, \Gamma_2 \rangle}{\Gamma \vdash e_1[\texttt{"indexOf"}](e_2) \Rightarrow \langle \texttt{number}, \Gamma_2 \rangle}\ (\textsc{SIndexOf})$$

$$\frac{\Gamma \vdash e \Rightarrow \langle \tau_1, \Gamma_1 \rangle}{\Gamma \vdash e\ \texttt{in}\ x \Rightarrow \langle \texttt{bool}, \Gamma_1[x:\texttt{object}] \rangle}\ (\textsc{In}) \qquad \frac{\Gamma \vdash e1 \Rightarrow \langle \tau_1, \Gamma_1 \rangle \quad \Gamma_1 \vdash e_2 \Rightarrow \langle \tau_2, \Gamma_2 \rangle}{\Gamma \vdash e_1 == e_2 \Rightarrow \langle \texttt{bool}, \Gamma_2 \rangle}\ (\textsc{Binop}) \qquad \frac{\Gamma \vdash e \Rightarrow \langle \texttt{string}, \Gamma_1 \rangle}{\Gamma \vdash e[\texttt{"length"}] \Rightarrow \langle \texttt{number}, \Gamma_1 \rangle}\ (\textsc{SLength})$$

Figure 2. Optional data-flow typing rules.

control breaks are optional. That is, from the typing point of view, it is enough to consider the function `h` equivalent to:

```
function h( x ) {
   if( x > 0 ) throw x;
   if( fake ) errno = 3;
}
```

To implement this solution, we extend the language as follows:

$$\text{statements} \quad s \quad ::= \quad \ldots \mid \textbf{break} \mid \textbf{continue} \mid \textbf{throw}\ e$$

and we add a new parameter to the typing rules. It accounts for the presence of breaks in the control flow. It is *true* if an instruction may break or return. It is *false* otherwise. Type judgments become $\Gamma \vdash inst \Rightarrow \langle \tau, \kappa, \Gamma' \rangle$. Figure 3 we add three rules for the new statements (Break, Cont, and Throw). We modify the ,Return rule as we no longer impose it to be terminal and we split the Seq rule in two. If the first statement of a sequence does not break, its typing is unchanged. If it breaks, the sequence is typed as a conditional instruction, meaning that the resulting typing environment is the merge of those of the two sub-statements (rule SeqBrk). Provided with these additional rules, the data-flow typing can cope with the full JavaScript language.

3.4 Wrap up

After the data-flow analysis completes, the compiler executes another traversal of the tree to assign precise types, *i.e.,* types that are neither $\top$ or $\bot$, to local variables and formal parameters for which the analysis has proved that a single type is preserved at all initialization and assignment locations. This yields a decorated AST of the program where types are added to variable declarations and variable references.

4 Hint Typing

The data-flow type analysis collects types for all occurrences of all variables and formal parameters but it looses track of values when: *i)* functions are exported, *ii)* functions are used as closures, *iii)* values are stored in data structures. The *hint typing* helps in these situations. It refines the inferred types, and restarts the whole typing process with these more precise pieces of information. A fix point iteration proceeds until no new type is collected.

The *hint typing* consists in traversing the program, scanning variables references in order to allocate *heuristic* types, or *hints*, to variable occurrences that the data-flow typing has not been able to determine precisely. Hints are assigned according to the syntactic contexts of the references and to the types already collected. Once hints have been collected, the most likely type of each yet untyped function parameter is elected and the function definition is duplicated for the specialized typed arguments. The initial calls for which the argument types match are replaced with calls to the specialized function.

Let us illustrate this principle with the `reverse` function below that reverses the elements of an array-like data structure. The example also defines two other functions, `areverse` that calls `reverse` with an array, and `oreverse` that calls it with an object. This polymorphic use confuses the data-flow analysis that merges the two types, `array` and `object`, into a single `any` type. The source code below shows the function definitions and the types inferred by the data-flow typing, expressed using the TypeScript syntax:

```
function reverse(a:any):any {
   for(let i:int = 0; i < a.length/2; i++) {
      let v = a[ a.length-1-i ];
      a[ a.length-1-i ] = a[ i ]; a[ i ] = v;
   }
```

$\boxed{\Gamma \vdash e \Rightarrow \langle \tau, \kappa, \Gamma' \rangle}$

$$\frac{}{\Gamma \vdash \textbf{break} \Rightarrow \langle \bot, true, \Gamma \rangle}\ (\textsc{Break}) \qquad \frac{}{\Gamma \vdash \textbf{continue} \Rightarrow \langle \bot, true, \Gamma \rangle}\ (\textsc{Cont}) \qquad \frac{\Gamma \vdash e \Rightarrow \langle \tau, \kappa, \Gamma_1 \rangle}{\Gamma \vdash \textbf{return}\ e \Rightarrow \langle \bot, true, \Gamma \rangle}\ (\textsc{Return})$$

$$\frac{\Gamma \vdash s_1 \Rightarrow \langle \tau_1, false, \Gamma_1 \rangle \quad \Gamma_1 \vdash s_2 \Rightarrow \langle \tau_2, \kappa_2, \Gamma_2 \rangle}{\Gamma \vdash s_1 ; s_2 \Rightarrow \langle \bot, \kappa_2, \Gamma_2 \rangle}\ (\textsc{Seq}) \qquad \frac{\Gamma \vdash s_1 \Rightarrow \langle \tau_1, true, \Gamma_1 \rangle \quad \Gamma_1 \vdash s_2 \Rightarrow \langle \tau_2, \kappa_2, \Gamma_2 \rangle}{\Gamma \vdash s_1 ; s_2 \Rightarrow \langle \bot, true, \Gamma_1 \sqcup \Gamma_2 \rangle}\ (\textsc{SeqBrk})$$

$$\frac{\Gamma \vdash e \Rightarrow \langle \tau, \kappa, \Gamma_1 \rangle}{\Gamma \vdash \textbf{throw}\ e \Rightarrow \langle \bot, true, \Gamma \rangle}\ (\textsc{Throw})$$

Figure 3. Typing rules with breaking control flow operators.

```
}
const areverse = ():any => reverse(a[23,56,3]);
const oreverse = ():any => reverse({length:2,"0":1,"1":45});
```

In the function `reverse`, the expression `a.length/2` reveals that the variable `a` is likely to be an array or a string, as these are the two widely used JavaScript primitive types that have a `length` property. This suspicion is strengthened by the expression `a[i]` that suggested an indexed access. Array type is even more suspected as strings are generally not accessed character by character but rather using API methods or regular expressions. The hypothesis of `a` being a string is definitively invalidated by the expression `a[i]=v` as JavaScript strings are immutable. Using that reasoning, the hint typing allows us to transform this program into:

```
function reverse$$A(a:array):array {
   for( let i:int = 0; i < aLen(a)/2; i++ ) {
      let v = aRef( a, aLen(a)-1-i);
      aSet(a, aLen(a)-1-i, aRef(a, i)); aSet(a, i, v);
   }
}
function reverse$$(a:any):any {
   for( let i:int = 0; i < a.length/2; i++ ) {
      let v = a[ a.length-1-i ];
      a[ a.length-1-i ] = a[ i ]; a[ i ] = v;
   }
}
const reverse = (a:any):any =>
      a instanceof Array ? reverse$$A(a) : reverse$$(a);
const areverse = ():array  => reverse$$A(a[ 23, 56, 3 ]);
const oreverse = ():any => reverse$$({length:2,"0":1,"1":45})
```

At the price of a code growth, the new code is faster because the array accesses have been specialized. This enables two additional optimizations: an array bound checking optimization (Section 5) and the inlining of array accesses (Section 6).

Hints and type likelihood are computed with rules of the form:

$$[\![inst_x]\!] \rightarrow \mathcal{H}(x, h_1, w_1), \ldots, \mathcal{H}(x, h_n, w_n)$$

which reads as follows: for each occurrence of the instruction $inst_x$ involving the variable x, x could have the type h_1 with weight w_1, ..., and type h_n with weight w_n. When all hints have been computed, a type likelihood is computed for all yet untyped formal parameters. The most likely type is selected and the function is duplicated accordingly. Two rules supplements this overly simple heuristic. In case of equally likely types, the ordering `array` < `string` < `object` applies. When a parameter is *i)* written in the function and *ii)* hinted as being potentially an object and either `null` or `undefined`, no specialization takes place because the compiler will not be able to use the parameter type annotation to generate better code.

Figure 4 shows a significant sample of the rules used by the compiler, where the notation $T(e)$ designates the type of e. The rules apply to a version of the language extended with binary operators, increment, and `switch` statements.

expr	e	::=	$\ldots \mid e$+$e \mid e$<$e \mid x$++
stmt	s	::=	$\ldots \mid$ **switch**(e) {**case** $e_1 : \ldots$**case**$e_n : \ldots$}

Rules 1, 2, and 3 handle property accesses. Rule 2 and 3 refine rule 1 by observing that when the property name is an integer or the string "`length`", the accessed data structure is likely to be an array or a string. Rules 3 and 4 handle property assignments. The weights are chosen to rule out the string type. The rules 6-9 are examples of the numerous rules that handle unary and binary operators. Rule 10 is more interesting. It says that if all the `case` expressions of a switch are of a certain type τ, the tested expression is then likely to be of that type τ. When two rules apply for the same expression, the hints and weights are summed up. For example, for the expression `x++ % y`, the two rules $\mathcal{H}(x, int, 1)$, $\mathcal{H}(y, int, 1)$ apply. Let us show how these rules apply to the `reverse` function.

$[\![$ `a["length"]` $]\!]$	by rule (1)	$\Rightarrow$	$\mathcal{H}(a,$ object$, 1)$
$[\![$ `a["length"]` $]\!]$	by rule (3)	$\Rightarrow$	$\mathcal{H}(a,$ object$, 1)$, $\mathcal{H}(a,$ array$, 2)$, $\mathcal{H}(a,$ string$, 2)$
$[\![$ `a[i]` $]\!]$	by rule (1)	$\Rightarrow$	$\mathcal{H}(a,$ object$, 2)$, $\mathcal{H}(a,$ array$, 2)$, $\mathcal{H}(a,$ string$, 2)$
$[\![$ `a[i]` $]\!]$	by rule (2)	$\Rightarrow$	$\mathcal{H}(a,$ object$, 2)$, $\mathcal{H}(a,$ array$, 4)$, $\mathcal{H}(a,$ string$, 4)$
$[\![$ `a[i]=v` $]\!]$	by rule (4)	$\Rightarrow$	$\mathcal{H}(a,$ object$, 4)$, $\mathcal{H}(a,$ array$, 4)$, $\mathcal{H}(a,$ string$, 4)$
$[\![$ `a[i]=v` $]\!]$	by rule (5)	$\Rightarrow$	$\mathcal{H}(a,$ object$, 4)$, $\mathcal{H}(a,$ array$, 7)$, $\mathcal{H}(a,$ string$, 4)$

...

This establishes that `array` is the most likely type for the variable `a` and the compiler will then generate efficient specialized code for that type.

$$
\begin{array}{lll}
(1) & [\![\ x[e]\ ]\!] & \rightarrow \quad \mathcal{H}(x, \texttt{object}, 1) \\
(2) & [\![\ x[e]\ ]\!] \wedge T(e) = \texttt{int} & \rightarrow \quad \mathcal{H}(x, \texttt{array}, 2), \mathcal{H}(x, \texttt{string}, 2) \\
(3) & [\![\ x[\texttt{"length"}]\ ]\!] & \rightarrow \quad \mathcal{H}(x, \texttt{array}, 2), \mathcal{H}(x, \texttt{string}, 2) \\
(4) & [\![\ x[e]{=}e\ ]\!] & \rightarrow \quad \mathcal{H}(x, \texttt{object}, 2) \\
(5) & [\![\ x[e]{=}e\ ]\!] \wedge T(e) = \texttt{int} & \rightarrow \quad \mathcal{H}(x, \texttt{array}, 3) \\
(6) & [\![\ x{<<}y\ ]\!] & \rightarrow \quad \mathcal{H}(x, \texttt{int}, 1), \mathcal{H}(y, \texttt{int}, 1) \\
(7) & [\![\ x\%y\ ]\!] & \rightarrow \quad \mathcal{H}(x, \texttt{int}, 1), \mathcal{H}(y, \texttt{int}, 1) \\
(8) & [\![\ x{+}e\ ]\!] \wedge T(e) = \texttt{num} & \rightarrow \quad \mathcal{H}(x, \texttt{num}, 1) \\
(9) & [\![\ x{+}{+}\ ]\!] & \rightarrow \quad \mathcal{H}(x, \texttt{int}, 1) \\
(10) & [\![\ \texttt{switch}(x)\{\texttt{case}\ e_1 : \ldots \texttt{case}\ e_n :\}\ ]\!] \wedge \forall i \in [1..n]\ T(e_i) = \tau & \rightarrow \quad \mathcal{H}(x, \tau, 1)
\end{array}
$$

Figure 4. Hint typing rules.

4.1 Conclusion and further comments

Once hints are computed and functions duplicated, the AST is cleaned up. The function specialization makes it possible to resolve statically some type checks and to remove dead-code. For some programs, it also happens that the generic function definition is never used and then removed from the tree. When these simplifications are applied, the whole typing process restarts. Each iteration improves the opportunities of discovering and refining new types. In the current version of the compiler, the granularity of the code duplication is the function but it might be worth investigating finer grain duplication, for instance for duplicating loops. Currently, speed is traded for size because functions are duplicated only once, which ensures the computation termination.

The hint typing stage delivers a decorated AST. Variable declarations and references hold more precise types than the data-flow analysis alone could have discovered. This AST is suitable for the last type analysis that follows. The hint typing is an incarnation of the assumption motivating this study: it is a tool the compiler uses to estimate the quality of the different versions it may generate for a same function.

5 Range Analysis

The *data-flow typing* and the *hint typing* work hand in hand to improve the precision of the types they collect but they are unable to produce refined annotations for numeric types. For that, the compiler relies on a *range analysis*. It is a central element toward good performances as the JavaScript specification exposes only double IEEE 754 numbers, whose performance do not compete with those of fix integer values. The range analysis annotates precisely the AST so that the code generator can map some numerical values into integer hardware registers and omit overflow checks.

The range analysis computes for each integer expression an approximation of the possible values it may evaluate to, represented as an interval. When the analysis completes, a tree traversal maps the general numerical types `integer` and `number` to precise types such as `index` (an integer in the range $[0, 2^{32}-2]$), `length` (*i.e.*, $[0, 2^{32}-1]$), `uint32`, etc. Applied to the `reverse$$A` function, the analysis establishes the following intervals:

```
function reverse$$A(a) {
   for(let i[0,0] = 0; i < aLen(a)/2[0,(2^32−1)/2]; i++[0,2^32−2]) {
      ...
   }
}
```

which enables the compiler to map the variable `i` to an `uint32` integer in the generated code. This also enables the expression `i++` to be executed without overflow detection and it enables fast array accesses, as shown in Section 6.

The range analysis handles only integer variables (for the data-flow typing and the hint typing, integer values are mere unbounded exact numbers, without range restriction). All other variables are considered as potentially holding infinite values, approximated with the interval $[-\infty, \infty]$.

The range analysis is based on RATA, a typed analyzer for JavaScript (Logozzo and Venter 2010) but it departs from the previous work by relying on a new technique for insuring convergence. For the sake of conciseness we briefly present main analysis, as it is fairly standard, and we focus on the convergence operator, usually named *widening operator* in the abstract interpretation community, as it is a crucial element of the overall quality of the analysis.

5.1 The Abstract Interpretation

The range analysis is presented in Figure 5 as a set of typing rules, based on those of Section 3 where expressions are extended to binary numerical operators and types are extended with integer intervals. In addition to the previous notations, $\triangledown$ is a widening operator (see Section 5.2) and we note $\Gamma \uplus \{x < n\}$ a new typing environment where the variable x is constraint to be smaller than the value n (see the IfRangeNum rule).

$$
\begin{array}{llll}
\text{expr} & e & ::= & \ldots \mid e{+}e \mid e{<}e \\
\text{type} & \delta & ::= & \ldots \mid [int, int]
\end{array}
$$

The critical part of the analysis is the definitions of the interval operations for binary and unary operators and for tests, as explicitly used in the Plus rule and implicitly in the IfRangeNum, and IfRangeVar rules. These operators govern the whole analysis by specifying how to compute approximations of integer operations. They are defined in figure 6.

$\boxed{\Gamma \vdash e \Rightarrow \langle \delta, \Gamma' \rangle}$

$$\frac{}{\Gamma \vdash n \;\Rightarrow\; \langle [n,n], \Gamma \rangle}\ (\textsc{Num}) \qquad \frac{x : \delta \in \Gamma}{\Gamma \vdash x \;\Rightarrow\; \langle \delta, \Gamma \rangle}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e_1 \;\Rightarrow\; \langle \delta_1, \Gamma_1 \rangle \quad \Gamma_1 \vdash e_2 \;\Rightarrow\; \langle \delta_2, \Gamma_2 \rangle}{\Gamma \vdash e_1 \texttt{ + } e_2 \;\Rightarrow\; \langle \triangledown(\delta_1 \oplus \delta_2), \Gamma_2 \rangle}\ (\textsc{Plus})$$

$$\frac{\Gamma \vdash e \;\Rightarrow\; \langle \delta, \Gamma_1 \rangle}{\Gamma \vdash x\texttt{=}e \;\Rightarrow\; \langle \delta, \Gamma_1[x:\delta] \rangle}\ (\textsc{Assign}) \qquad \frac{\Gamma \vdash s_1 \;\Rightarrow\; \langle \delta_1, \Gamma_1 \rangle \quad \Gamma_1 \vdash s_2 \;\Rightarrow\; \langle \delta_2, \Gamma_2 \rangle}{\Gamma \vdash s_1 \texttt{;} s_2 \;\Rightarrow\; \langle \bot, \Gamma_2 \rangle}\ (\textsc{Seq}) \qquad \frac{\Gamma \sqcup \Gamma_2 \vdash e \;\Rightarrow\; \langle \delta, \Gamma_1 \rangle \quad \Gamma_1 \vdash s \;\Rightarrow\; \langle \bot, \Gamma_2 \rangle}{\Gamma \vdash \texttt{while(}e\texttt{) \{}s\texttt{\}} \;\Rightarrow\; \langle \bot, \Gamma_1 \sqcup \Gamma_2 \rangle}\ (\textsc{While})$$

$$\frac{\Gamma \vdash e \;\Rightarrow\; \langle \delta, \Gamma_1 \rangle \quad \Gamma_1 \vdash s_1 \;\Rightarrow\; \langle \bot, \Gamma_2 \rangle \quad \Gamma_1 \vdash s_2 \;\Rightarrow\; \langle \bot, \Gamma_3 \rangle}{\Gamma \vdash \texttt{if(}e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \;\Rightarrow\; \langle \bot, \Gamma_2 \sqcup \Gamma_3 \rangle}\ (\textsc{If}) \qquad \frac{\Gamma_1 \uplus \{x < n\} \vdash s_1 \;\Rightarrow\; \langle \bot, \Gamma_2 \rangle \quad \Gamma_1 \uplus \{x \geq n\} \vdash s_2 \;\Rightarrow\; \langle \bot, \Gamma_3 \rangle}{\Gamma \vdash \texttt{if(}x < n\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \;\Rightarrow\; \langle \bot, \Gamma_2 \sqcup \Gamma_3 \rangle}\ (\textsc{IfRangeNum})$$

$$\frac{x : [x_l, x_h],\, y : [y_l, y_h] \quad \Gamma_1 \uplus \{x < y_l\} \uplus \{y > x_h\} \vdash s_1 \;\Rightarrow\; \langle \bot, \Gamma_2 \rangle \quad \Gamma_1 \uplus \{x \geq y_l\} \uplus \{y \leq x_h\} \vdash s_2 \;\Rightarrow\; \langle \bot, \Gamma_3 \rangle}{\Gamma \vdash \texttt{if(}x < y\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \;\Rightarrow\; \langle \bot, \Gamma_2 \sqcup \Gamma_3 \rangle}\ (\textsc{IfRangeVar})$$

Figure 5. Interval analysis.

Let us consider the following conditional expression: `if(x < y)` *then* `else` *else*, and let us assume that `x` and `y` are known to be in the intervals $[x_l, x_u]$ and $[y_l, y_u]$. Some knowledge can be deduced in both branches. In the *then* branch, `x` is known to be smaller than `y`, which potentially narrows its approximation. The value `x` may hold is the interval obtained by computing $[x_l, x_u] \prec [y_l, y_u]$. Interestingly, in the *then* branch, the test also narrows the approximation of `y`, as it is known to be greater or equal to `x`. The same reasoning applies to the *else* branch, where `x` is known to be in the interval $[x_l, x_u] \geq [y_l, y_u]$ and `y` in $[y_l, y_u] \leq [x_l, x_u]$.

5.2 Widening and Stepping

The range analysis relies on a widening operator to ensure its convergence in an acceptable compilation time. For instance, for the `for` loop of the `reverse$$A` function, it enables to compute the final approximation interval $[0, 2^{32}-2]$ of the variable `i` in less than $2^{32}-2$ steps! For that, instead of adding 1 to `i` at each iteration as the execution of the program will do, the analysis adds larger and larger integer values. In the range analysis, each time the instruction is analyzed, a larger than before value is added. This is designated as a *delaying strategy* in (Cousot et al. 2007).

The widening takes place after each abstract interpretation of a numeric operation. Let us illustrate its principle with the `i` increment. Let us assume that at one moment of the analysis, the variable `i` is approximated by the interval $[l, u]$. The constant is interpreted as $[1, 1]$ and the interpretation of the addition produces the interval $[l+1, u+1]$. This interval is then widened into $[m, v]$ with $m \leq l$ and $v \geq u+1$. Following the conventions of the domain, we note $[m, v] = \triangledown[l+1, u+1]$.

The widening operator we use relies on numerical scales and a stepping process. Intervals are widened progressively, that is step by step, using two different scales for intervals lower and upper bounds. These scales are established based on the JavaScript specification and on some remarkable integer values many programs use.

The JavaScript specification makes use of some *special* integers. First, as numbers correspond to a double-precision 64-bit binary format IEEE 754 values, integers are restricted to the interval $[-2^{53}, 2^{53}]$. Second, JavaScript defines array lengths as integers in the range $[0, 2^{32}-1]$, which implies that the largest array index is smaller or equal to $2^{32}-2$. These integer values are included in our widening scale. We also add a few numbers of our own. Hopc's backend uses two-bit tagged integers, so the largest integer value on a 32 bit machine is $2^{30}-1$. We include that value in our widening scale and for the negative values, we include -1 and -2, as these numbers are frequently used for terminating decreasing loops. In conclusion, we use the following scales:

upper bound steps: $0, int30, index, length, int53, +\infty$
lower bound steps: $0, -1, -2, -int30, -int53, -\infty$

With the following notations: $int30 = 2^{30}-1$. $index = 2^{32}-2$, $length = 2^{32}-1$, and $int53 = 2^{53}$.

We can now complete the explanation of the result of the range abstract interpretation for the `reverse$A` function. The operator `aLen` returns an array length, then `aLen(a)` $\in [0, length]$. We derive `aLen(a)/2` $\in [0, length/2]$. At each iteration of the loop, the variable `i` is incremented and the interval widened. It is then successively approximated by $[0, 0]$, $[0, int30]$, and ends with $[0, index]$, $index$ being the smallest value of the upper bound scale greater than $length/2$.

5.3 Final word

Once the range analysis completes, the intervals are used to assign precise integer types to expressions and variable declarations. These types are used to improve the performance of the generated code. Obviously, they enable type checks removal but even more importantly, they enable numbers to be untagged and unboxed. This is presented in the next section.

6 Implementation

Fast JavaScript compilers go beyond implementing well a small core language. They also deploy a large arsenal of complex optimizations and runtime techniques. Hopc implements some of them. Describing all of Hopc is beyond the scope of this paper. Here, we only focus on the type analyses it performs and how it is used to shape the generated code. This is illustrated by the code generated for iterating over

addition	$[l_1, u_1] \oplus [l_2, u_2]$	=	$[l_1 + l_2, u_1 + u_2]$
substraction	$[l_1, u_1] \ominus [l_2, u_2]$	=	$[l_1 - u_2, u_1 - l_2]$
multiplication	$[l_1, u_1] \otimes [l_2, u_2]$	=	$[min(l_1 * l_2, u_1 * u_2, l_1 * u_2, l_2 * u_1), max(l_1 * l_2, u_1 * u_2, l_1 * u_2, l_2 * u_1)]$
division	$[l_1, u_1] \oslash [l_2, u_2]$	=	$[trunc(l_1/u_2), ceil(u_1/l_2)]$
smaller than	$[l_1, u_1] < [l_2, u_2]$	=	$if\ u_2 \leq u1\ then\ (if\ u_2 > l1\ then\ [min(l1, u_2 - 1), u_2 - 1]\ else\ [l_1, l1])\ else\ [l_1, u_1]$
smaller or equal	$[l_1, u_1] \leq [l_2, u_2]$	=	$if\ u_2 < u1\ then\ (if\ u_2 \geq l1\ then\ [min(l1, u_2), u_2]\ else\ [l_1, l1])\ else\ [l_1, u_1]$

Figure 6. Interval operators.

arrays, which is a recurrent JavaScript programming pattern as arrays are ubiquitous in this language.

Remember that JavaScript primitive numbers are double-precision 64-bit IEEE 754 values but array indexes and bit-wise operations are specified over 32-bit fix integers. Using ad-hoc representations that fit hardware integer registers for these numbers is crucial for performances. Hopc uses the results of the previous type analyses (occurrence typing, hint typing, and range analysis) to use the most efficient number representation, expression by expression. That is, as much as possible, it generates code that uses native unsigned 32-bit integers for representing indexes, native signed 32-bit integers when values can be negative, and polymorphic representations that use tagged integers and boxed floating point numbers otherwise. Let us consider the following example, compiled for 32-bit platforms:

```
let i = 0; while( i < a.length ) sum += a[ i++ ];
```

and let us assume that the hint typing has specialized the code for `a` being an array. The occurrence typing proves that `i` is an integer and the range analysis proves that it is in the range $[0..2^{32} - 2]$ (because of the JavaScript length specification). The variable `i` can then be mapped to a hardware register and the increment can be implemented as a simple assembly instruction with no type check, no tagging/untagging, and no overflow check. This is optimal but what happens now if we make `i` polymorphic by assigning it a value of a different type as in:

```
let i = 0; while( i < a.length ) sum += a[ i++ ];
i = null;
```

At the point of the increment, `i` is still known to be an integer in the range $[0..2^{32} - 2]$ but the variable is now polymorphic as it holds integers *and* the `null` value. So, it can no longer be represented as a native unsigned 32-bit integer. Its initial value is compiled as a polymorphic value: a tagged integers where the two lower bits are used to encode the integer type and the 30 higher bits are used to encode the actual integer value. 30-bit values are not large enough to encode all possible array indexes so the loop increment may overflow. This must be tested. After the increment `i` may either be a tagged 30-bit integer or a boxed double precision number. An additional test is then also needed before each increment to check which representation is used and to select the proper addition operator. On a modern 32-bit platform, we measured a factor of three between the execution times of the two versions. Avoiding polymorphic representations as much as possible is, performance wise, essential.

6.1 Integer Boxing/Unboxing

The first step of the untagging algorithm consists in computing for each integer variable v (see Section 5), $R_m(v) = [L..U]$, the smallest range that is larger than all the ranges in $R(v)$, the set of the ranges of all the v occurrences. $R_m(v)$ is the smallest range that verifies $\forall [l..u] \in R(v), L \leqslant l \wedge u \leqslant U$. The second step associates precise types to all expressions, using the following mapping:

(1)	$R(v) \in [0..2^{32} - 1]$	$\mapsto$	`uint32`	untagged 32-bit value
(2)	$R(v) \in [-2^{31}..2^{31} - 1]$	$\mapsto$	`int32`	untagged 32-bit value
(3)	$otherwise$	$\mapsto$	`integer`	tagged value

A variable reference type might be more specific than its declaration type, as in the example of the introduction. In the loop, the variable `i` is known to be an `uint32`. It is declared as an `uint32` in the first version, but it is declared as an `any` value in the second because of the `null` assignment.

The third step of the algorithm consists in inserting type coercions to switch from native representations to polymorphic representations and vice-versa. Values are tagged or boxed when: *i)* they are stored in objects and arrays, *ii)* they are stored in polymorphic variables, *iii)* they are arguments to untyped or polymorphic function calls, *iv* they are mutable and captured in a closure. They are untagged/unboxed in the converse operations.

6.2 Arrays

The combined use of the occurrence typing, hint typing, range analysis, and numbers untagging enables Hopc to map JavaScript numbers to 32-bit integers and to map simple operations such as unary operators, binary operators, and array accesses to simple assembly instructions. This is illustrated in this Section where it is studied how Hopc compiles loops over arrays, which is challenging because of sparse arrays and because arrays may dynamically grow and shrink. According to the *optimistic assumption* presented in Section 2, the compiler favors flat and non-extended arrays, which enables to generate efficient code for common situations.

The fast access of an object property relies on the hidden classes technique (Deutsch and Schiffman 1984). This is efficient for objects but this does not fit well arrays that are

accessed via integers instead of named properties. Hopc uses another schema that favors fast accesses inside loops. It supports efficiently arrays that are flat and that are only accessed via numerical properties.

Arrays are implemented as objects with 4 fields: a `properties` list for non numerical properties and for sparse array properties; a `length` that denotes the total number of numerical elements (only those that are indexed by an integer in the interval $[0, 2^{32} - 2]$); an `ilength` that denotes the number of elements of flat arrays; and a raw `vector` that contains the elements of flat arrays. Arrays are created flat, with an empty `properties` list. Arrays are *un-flattened* when an element is removed or when a non contiguous element is added. Generally, arrays remain flat during their whole lifetime. When executing a loop as:

```
function sum( a ) {
   let i = 0, sum = 0;
   while( i < a.length ) sum += a[ i++ ];
   return sum;
}
```

the values of the attributes `ilength` and `vector` are unlikely to change. So, they can be preloaded before the loop and used inside, where a mere guard checking that `i` is smaller than `ilength` is enough. If during the loop, `a` changes, either because an element is removed or added or because its length is modified, the `ilength` property is modified accordingly, which impacts the result of the guard for the next iteration. Using C's syntax, and eliding slightly, the generated code is equivalent to:

```
obj_t sum( obj_t a ) {
   uint32_t i, ilen = a->ilength;
   obj_t *v = a->vector, sum = JS_INT( 0 );

   for( i = 0; i < a->length; i++ ) {
      if( i < ilen ) { // fast path, flat array
         sum = JS_ADD( sum, v[ i ] );
      } else {         // slow path, complex array, update needed
         sum = JS_ADD( sum, JS_GET_PROPERTY( a, i ) );
         v = a->vector; ilen = a->ilength;
      }
   }
   return sum;
}
```

Using unboxed representations for `i` and its increment, as suggested in Section 6.1, the generated code loop is almost as fast as an equivalent C loop. It only imposes a mere additional comparison between two registers, one for the index that varies during the loop, and the `ilength` array field. Notice however that this almost optimal compilation only applies when the loop involves no code that could potentially delete array elements. In particular, the loop must not contain calls to unknown functions. When this cannot be established by the compiler only one an extra guard is needed before each access inside the loop. The performance evaluation presented in Section 7 shows how well this principle applies.

The combined use of the occurrence typing, hint typing, range analysis, and numbers untagging is a central element of the compilation process as it enables Hopc to map JavaScript numbers to 32-bit integers and to map simple operations such as unary operators, binary operators, and array accesses to simple assembly instructions.

7 Performance Evaluation

We have compared Hopc's performance and other JavaScript implementations, namely: Google's V8 (6.2.414.54), Rhino (1.7.7) the first historical AOT JavaScript compiler that generates JVM byte-code, JJS (9.0.4) the Adobe JavaScript compiler that generates JVM byte-code too, and JerryScript (1.0 c3c0bb8d), a JavaScript interpreter designed for IoT devices.

We have collected the execution times of popular JavaScript tests coming from the Octane, SunSpider, and JetStream test suites. From these test sets, we have ruled out floating point intensive programs because Hopc does not optimize floating point numbers yet and then all these tests are dominated by the garbage collection time. Hopc uses a conservative Mark&Sweep garbage collector (Boehm and Weiser 1988), which is a technique known not to be efficient for handling short living objects. Second we have eliminated browser-only programs for obvious reasons. And finally, we have also eliminated very large tests (larger than 10.000 lines of code in a single file) as our compiler, which is meant for separate compilation, cannot cope with such large source files (compilation times become excessively long, up to 30 minutes, on very large source files). Each program has been executed 30 times and we have computed the median and the deviation of the execution wall clock times. Figure 7 presents these results relatively to V8 performance that establishes the baseline of our comparison. Benchmarks where executed unmodified but, when possible, the number of iterations was configured so that a test runs in no less than 10 seconds.

Unsurprisingly JerryScript, the sole interpreter of our experiment, is in between one and two orders of magnitude slower than compilers. This system has being designed for running on tiny devices, it is optimized for space, not for speed, contrarily to compilers that use the opportunity to generate several versions of the same source code and to use various memory caches to run faster.

On many tests V8 and Hopc are in the same range, separated by a factor of 2 or 3. In the best situations, Hopc outperforms V8 significantly (`base64`, `fib`, `qsort`, and `splay`). These are integer and array intensive programs that fully benefit from the type analyses and tagging/untagging optimization presented in Section 6. The tests `crypto`, `crypto-aes`, `crypto-md5` are actually disguised floating point numbers test. They perform many bitwise operations on 32-bit integers and store them into arrays. On 32-bit platforms, Hopc represents these integers as floating point numbers and allocate

	jit	*AOT compilers*			*interpreter*
benchmark	V8	Hopc	jjs	Rhino	JerryScript
bague★	1.00 4.6%	0.99 0.5%	_	20.29 2.0%	142.50 0.0%
base64°	1.00 0.9%	0.59 3.7%	2.00 1.9%	7.07 0.7%	_
boyer†	1.00 1.1%	1.47 1.1%	8.10 24.9%	7.44 3.7%	_
crypto†	1.00 0.4%	7.31 0.3%	2.97 6.8%	25.60 8.4%	116.54 0.0%
crypto-aes*	1.00 0.4%	2.34 0.3%	4.56 2.4%	11.68 4.8%	86.88 0.0%
crypto-md5*	1.00 2.7%	6.51 0.3%	2.82 8.8%	7.69 2.5%	54.54 0.0%
deltablue†	1.00 0.4%	6.81 0.2%	6.78 4.2%	54.37 4.7%	290.85 0.0%
fannkuch◇	1.00 0.1%	1.70 0.4%	1.80 19.6%	5.89 2.2%	98.10 0.0%
fib★	1.00 0.3%	0.66 0.2%	1.66 1.2%	2.76 0.6%	50.57 0.0%
maze★	1.00 0.6%	1.20 2.9%	_	_	_
puzzle★	1.00 1.7%	1.89 0.2%	2.17 1.2%	6.23 0.8%	_
qsort★	1.00 0.1%	0.91 0.2%	1.47 1.1%	_	_
richards†	1.00 0.3%	3.34 0.4%	2.59 2.0%	24.64 1.3%	156.71 0.0%
sieve★	1.00 0.3%	5.02 0.7%	_	_	_
splay†	1.00 16.0%	1.17 0.8%	2.59 5.9%	5.27 1.3%	_
tagcloud*	1.00 1.0%	3.45 0.8%	1.99 7.8%	5.80 5.9%	_

Figure 7. Results of 30 runs collected on an Intel core i7-3520M running Linux4.13/Debian configured for 32-bit executions. Median of wall clock times relative to V8 performance and deviations divided by the mean. Smaller time is better. Benchmark sources: † Octane, * Jetstream, ° Sunspider, ★ Bglstone, ◇ other sources.

them. The execution time is then dominated by the garbage collection (henceforth GC) time that represents more than 65% of the overall execution time and by double precision operations that count for 15% of the execution.

The test `deltablue` performs poorly with Hopc compared to V8. It is an allocation intensive programs whose execution time is dominated by allocations of short living objects, which is an allocation pattern the garbage collector does not handle efficiently. Improving on that aspect, is certainly a subject for further studies (Blackburn and McKinley 2008).

The test `tagcloud` uses the `eval` function to create a large data structure. Independently of the significant execution time spent in the interpreter, this shows that even in the presence of direct `eval` in the source, Hopc is still able to generate decently efficient code.

7.1 Hint Typing Performance

To evaluate the hint typing impact we have instrumented the code generator to collect statistics about function invocations (Figure 8). The compiler instruments the generated code to mark function calls with one of the following tag: typed, untyped, hinted, unhinted, and dispatch. By comparing the *hinted calls*, *unhinted calls*, and *dispatch calls* numbers we can measure the effectiveness of the function specialization.

The first observation is that for all tests where it applies but `crypto-md5`, the hint typing successfully specializes function definitions. For some benchmarks such as `maze` the specialized functions are even invoked directly without going through a dynamic dispatch. This optimal situation happens when the data-flow analysis or the range analysis discover that for a particular call site the specialized function call be called directly. Other tests such `crypto`, `sieve`, or `splay` use the dispatch function. This correspond to situations where the type analyses alone are not able to discover sufficiently precise types.

The hint typing gives poor results for the `crypto-md5`. It is the only test that counts an important number of *unhinted* calls. This benchmark uses 32-bitwise operations extensively that the hint typing successfully specializes. For instance, for the function `md5_ff` defined as

```
function md5_ff(a, b, c, d, x, s, t) {
  return md5_cmn((b & c) | ((~b) & d), a, b, x, s, t);
}
```

the arguments `b`, `c`, and `d` are correctly specialized as `int32` integers. However, this test also uses 32-bit literal constants which can only be represented as floating point numbers on a 32-bit platform. This causes type miss-matches between the specialized functions and their actual parameters. Note that this problem disappears on 64 bit platforms where 32-bit are represented as exact integers.

This experiment shows that although simple, the rules presented in Section 4 are sufficient to guess correctly runtime program behaviors and that there is no need to invent more complex analyses to discover the best typing context for a function definition.

8 Related Work

It is frequent that JavaScript variables are assigned values of different types and that functions are not called with the declared number of arguments. Thus, the typing approaches

benchmark	*typed calls*	*untyped calls*	*hinted calls*	*unhinted calls*	*dispatch calls*
bague	1647×10^9	11	0	0	0
base64	22	0	0	0	0
boyer	400	143×10^6	401	1×10^3	1×10^3
crypto	14×10^6	3×10^6	46×10^6	165×10^3	46×10^6
crypto-aes	59×10^6	510×10^3	0	0	0
crypto-md5	126×10^6	0	161×10^6	160×10^6	128×10^6
deltablue	80×10^3	82×10^6	80×10^3	80×10^3	1
fannkuch	11	0	0	0	0
fib	-206752951	0	0	0	0
maze	38×10^6	80×10^6	1×10^6	0	1×10^6
puzzle	39×10^6	61	0	0	0
qsort	66×10^6	39×10^6	0	0	0
richards	0	902×10^3	1	0	1
sieve	66×10^6	281×10^6	270×10^6	352×10^3	271×10^6
splay	0	2×10^6	71×10^6	0	71×10^6
tagcloud	0	0	2×10^3	0	2×10^3

Figure 8. Statistics of function invocations. Typed calls correspond to functions successfully typed by the data-flow and range type analyses. Hinted calls correspond to functions typed by the hint typing. Unhinted calls are functions invocations for which the specialized version has not been selected at runtime. Dispatch calls are the number of hinted function invocations that need a runtime type check.

that assign unique types to variable declarations (Anderson and *et al.* 2005; Lerner et al. 2013) are mildly effective for real-life JavaScript programs. The code specialization enabled by the hint typing is not affected by this problem as it chooses the types according to variable uses, not only variable declarations.

The flow analysis presented in Section 3 follows a line of research that uses abstract interpretation techniques for assigning types to expressions (Jensen et al. 2009). In the past, these analyses have been mostly used for designing programming environment tools rather than included in a compilation process. This is the objective of the hybrid type inference (Hackett and Guo 2012). It consists in a static type analysis designed for producing type information used at runtime by a JIT compiler. It shares many similarities with our approach, in particular because the static analysis is unsound and seconded by runtime guards. The system maintains a dichotomy between static *may-have-type* and *must-have-type* and types which could potentially be observed and types that have already been observed at runtime. We do not provide anything similar but we might accommodate this idea in the future. This study also mentions an integer overflow detection but does not give any details. It seems relatively simple and less precise than our range analysis.

The data-flow type analysis is an *occurrence typing* analysis tailored for hint typing. It is simpler than the original occurrence typing developed for the Scheme programming language (Kent et al. 2016; Tobin-Hochstadt and Felleisen 2010) as it only handles simple types and simple type checks. Although this seems precise enough for the needs of the hint typing, it might be worth incorporating more precise analyses in the future.

There is a whole line of research on JavaScript static analysis. The main systems are TAJS (Andreasen and Møller 2014), Wala (Schäfer et al. 2013), and Safe (Park and Ryu 2015). These systems rely on complex abstract interpretations. They are able to deduce accurate information about programs, but their complexity and execution times make them unusable in practical compilers.

The range analysis presented in Section 5 departs from RATA, a typed analyzer for JavaScript (Logozzo and Venter 2010). First, our analysis uses the type information collected by previous compilation stages. Second, we consider a different arithmetic lattice as we target 32 bit machines for which array length cannot be represented as 32 bit integers, if tagging is used. Then, we consider `uint30`/`uint32` for tagged/untagged representations, which have to be included in the analysis. Last, the threshold we use for the delayed widening is different. We do not collect the constants found in the program as this cannot cope with programs where the loop upper bounds are computed values, for instance, an array length. Instead we use a static scale based on pre-defined values.

In the seminal description of polymorphic inline cache (Hölzle et al. 1991) the authors mention a *type prediction* mechanism used in SELF and Smalltalk compilers. In a 7-line long paragraph they mention that the compilers predict that the argument to `+` are predicted to be integer. This obviously relates to the hint typing but the lack of details of their presentation makes is hard to compare the two approaches.

Samsung's Sjs (Choi et al. 2015) is an AOT JavaScript compiler. It relies on a type system and a type inference that

provide information that the rest of the compilation chain uses to generate efficient code. A recent paper (Chandra et al. 2016) reports excellent execution times comparable to those of V8 (Google 2018) but also much smaller memory occupations. These execution times are better than those we have reported in Section 7, but this is mitigated by Sjs not considering full-fledged JavaScript as V8 or our compiler do. Sjs limits properties polymorphism, it does not comply with JavaScript prototype semantics, and more importantly, it seems not to support introspection and dynamic features such as computed field names and field deletions. Sjs cannot run the standard JavaScript unmodified. This is why it is not reported in Section 7. It is also unclear if it supports separate compilation that we need for accommodating the NPM module systems. These restrictions enable the type system to report precise information and of course simplify the code generation. They are probably imposed by the nature of the type inference algorithm. Our approach does not suffer from this limitation. However, it remains that Sjs is an excellent preliminary result and a strong incentive for pursuing investigations on the JavaScript static compilation. It is also a sensitive approach as Sjs is designed for enabling embedded JavaScript, a context in which programs are small, closed, and where it is probably fine not to support all the language features.

Bolz *et al.* have proposed *storage strategies* (Bolz et al. 2013) for optimizing the representations of homogeneously typed collections. It consists in associating each collection with an ad-hoc strategy that evolves over time when elements are added. This mechanism saves memory space and speeds up data accesses. Experimental results show significant benefits for the Python programming language. Clifford and his colleagues have developed analog solutions for JavaScript (D. et al. 2015). They have modified the V8 JIT compiler to cope with various storage representations for objects and arrays. They combine homogeneous data representations for fast storage and access and an allocation logging mechanism that works hand in hand with the garbage collector to avoid allocating extra unused space for objects. These previous studies focus on the efficiency of data representations. The fast array access array we have presented focuses on the efficiency of the loop control and data flows. They are different, they do not follow the very same goal, but they are complementary. Combining both could yield to fast accesses, fast loops, and efficient data storage.

9 Conclusion

This paper presents Hopc, a new AOT compiler for JavaScript. It relies on the observation that amongst all the possible interpretations a JavaScript program may have, the most likely is the one for which the compiler can deliver its best code. We have derived this principle in a type analysis called *hint typing* and in a code generator that uses flat untyped number representations. We have implemented the hint typing and the other analyses and optimizations it enables, namely a range analysis and untagging optimization.

The experimental report shows that Hopc approaches the performance of the fast JIT compilers on several benchmarks. Even if Hopc is still usually slower we think that this experiment establishes that AOT compilation is a promising approach for implementing languages as dynamic as JavaScript, especially in application domains such as IoT where many devices cannot use JIT compilers, either because they have too limited capacities or because they only support executable read-only memory, which makes JIT compilation unusable.

References

C. Anderson and *et al.* 2005. Towards Type Inference for Javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Heidelberg.

E. Andreasen and A. Møller. 2014. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*.

Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. New York, NY, USA.

H.J. Boehm and M. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software, Practice, and Experience* 18, 9 (Sept. 1988).

C. F. Bolz, L. Diekmann, and L. Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*.

C. Chambers and D. Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *Conference Proceedings on Programming Language Design and Implementation (PLDI '89)*. ACM, New York, NY, USA.

S. Chandra, C. S. Gordon, J-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi. 2016. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA.

M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. 2009. Tracing for web 3.0: trace compilation for the next generation web applications. In *In Proceedings of the International Conference on Virtual Execution Environments*.

M. Chevalier-Boisvert and M. Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*.

M. Chevalier-Boisvert and M. Feeley. 2016. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*.

W. Choi, S. Chandra, G. Necula, and L. Sen. 2015. SJS: A Type System for JavaScript with Fixed Object Layout. In *Static Analysis - 22nd International Symposium, SAS'15*. Saint-Malo, France, 181–198.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2007. *Combination of Abstractions in the ASTRÉE Static Analyzer*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Clifford. D., H. Payer, M. Stanton, and B. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*. New York, NY, USA.

P. Deutsch and A. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, NY, USA.

ECMA International. 2015. *Standard ECMA-262 - ECMAScript Language Specification* (6.0 ed.).

ECMA International. 2016. *ECMAScript Test Suite* (2 ed.). Technical Report TR/104.

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*.

A. Ghuloum and K. Dybvig. 2009. Fixing Letrec (reloaded). In *Workshop on Scheme and Functional Programming*. Cambridge, MA, USA.

L. Gong, M. Pradel, and K. Sen. 2014. *JITPROF: Pinpointing JIT-unfriendly JavaScript Code*. Technical Report UCB/EECS-2014-144.

Google. 2018. V8 JavaScript Engine. http://developers.google.com/v8.

B. Hackett and S-Y. Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA.

U. Hölzle, C. Chambers, and D. Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK.

S H. Jensen, A. Møller, and P. Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS '09)*. Springer-Verlag, Berlin, Heidelberg.

Kangax. 2018. ECMAScript Compatibility Table. https://kangax.github.io/compat-table/es6/.

A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. 2016. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. USA.

B. S. Lerner, J. Politz, J.G., A. Guha, and S. Krishnamurthi. 2013. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS '13)*. ACM, NY, USA.

F. Logozzo and H. Venter. 2010. RATA: Rapid Atomic Type Analysis by Abstract Interpretation - Application to JavaScript Optimization. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*.

Microsoft. 2013. TypeSscript, Language Specification, version 0.9.5.

C. Park and S. Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, Prague, Czech Republic*.

M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. 2013. Dynamic determinacy analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*.

S. Tobin-Hochstadt and M. Felleisen. 2010. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA.

O. Waddell, D. Sarkar, and K. Dybvig. 2005. Fixing Letrec: A Faithful Yet Efficient Implementation of Scheme's Recursive Binding Construct. *Higher-Order and Symbolic Computation* 18, 3 (2005).