Typed JS: A Lightweight Typed JavaScript Engine for Mobile Devices

Ryan H. Choi(✉) and Youngil Choi

Software R&D Center, Samsung Electronics, Suwon, South Korea
{ryan.choi,duddlf.choi}@samsung.com

Abstract. Web applications have been gaining huge popularity due to being platform independent and also enabling fast development. Unfortunately, due to insufficient performance of web applications, they are generally limited to non-performance-critical use. The performance of web applications is largely affected by the performance of JavaScript. To address this problem, modern JavaScript engines such as Google's V8 incorporate many state-of-the-art optimization and engineering techniques. In industry, recent approaches are to extend JavaScript to decorate objects with types to better utilize just-in-time (JIT) compilers.

In this paper, we present Typed JS, a subset of JavaScript that utilizes type-decorated syntax. Unlike previous approaches, Typed JS supports most of the JS core operations while utilizing the ahead-of-time (AOT) compilation technique, which was not possible in the existing solution. Typed JS is specifically designed for running Web applications on mobile devices with goals of having smaller memory footprint while achieving high-performance, which is accomplished by utilizing the type information and AOT technique. Experiments show that Typed JS requires significantly much less memory usage while performing better than industry-leading JavaScript engines on a mobile platform.

Keywords: Typed JavaScript · Static type · Mobile

1 Introduction

JavaScript is the standard Web programming language, commonly integrated to web browsers to allow users interact on client-side applications. Also, together with HTML and CSS, it defines the standard Web application framework, which allows developers to create large-scale and complex Web-based applications. Some popular web applications include Gmail, Google Docs, Facebook, etc. In these applications, JavaScript is typically used to execute complex user interaction and business logic. The strength of Web applications is, unlike traditional applications, it does not have platform dependency—any platform including desktop and mobile that includes a modern web browser can execute web applications. Furthermore, Web applications are self-maintainable from user's view in a way that users do not worry about installing and updating web applications.

S. Sigg et al. (Eds.): MobiCASE 2015, LNICST 162, pp. 107–121, 2015.
DOI: 10.1007/978-3-319-29003-4_7

Due to significant influence of JavaScript in the Web framework, recently, improving the performance of JavaScript has received much attention from both industry and academia. Modern industry-leading JavaScript engines including V8 [9] and JSC [2] have been heavily optimized over the last decade. Some well known optimization techniques include JIT (just-in-time), hidden class and inline cache. These optimization techniques are to address one of the fundamental designs of JavaScript—that is, JavaScript is a dynamic language such that, a JavaScript's object layout is unknown during the JavaScript compilation phase, but gradually known during runtime. For each object, information about the object layout is collected while executing JavaScript, and when the object layout is hardly altered, these optimization techniques start to optimize execution steps by generating machine code for faster execution (i.e., JIT) and caching properties (i.e., inline cache). A key concept of these techniques is that, object layouts do not change much after JavaScript executes for a while. Hence, one drawback is that, when object layout changes frequently, JIT and inline cache are not as effective. Furthermore, techniques such as JIT is a resource-intensive technique, so it may not be suitable for platforms with limited CPU and memory resources such as mobile platforms.

JavaScript, being dynamic by nature, presents many performance and memory optimization challenges. To address these problems, recently, a few variants of JavaScript are proposed, and being actively developed in industry. One common goal among these variants is to restrict JavaScript's dynamicity without much affecting JavaScript's design by adding static types. Objects declared with static types are not allowed to change types during runtime. Hence, by utilizing these extra type information, JIT and inline cache can be more effective, and more aggressive optimization techniques can be applied. TypeScript [14] is one of the first attempt in this direction. It extends JavaScript to accept type-decorated syntax. Flow [8] is a subset of JavaScript that also accepts type-decorated syntax. Unlike TypeScript, Flow is a type-checker such that, it analyzes type-decoration for consistency and correctness, and relies an existing JavaScript engine for code execution. They both attempt to fully utilize current optimization techniques such as JIT already implemented in JavaScript virtual machines (VMs) to maximize the performance on desktop by supplying type information and not modifying object layouts.

Ahead-of-time (AOT) compilation technique, unlike JIT optimization, requires all object types to be known during the code compilation phase. By fully understanding object types and not allowing changing object layouts during runtime, it can aggressively optimize for maximum performance. But due to not knowing object types and dynamicity in objects, AOT is not suitable for JavaScript. However, there is an attempt to integrate the strength of AOT compilation into JavaScript virtual machines. asm.js [15] translates C++ code into non-dynamic JavaScript code to make it execute faster than typical JavaScript code with some forms of dynamicity. Unfortunately, functionality of asm.js is limited in a way that, it cannot represent JavaScript core design such as objects, prototype, etc. Nevertheless, it proves that, eliminating dynamic behaviors can significantly increase the performance.

In this paper, we propose a design of a subset of JavaScript called Typed JavaScript (Typed JS), which utilizes type-decorated syntax, and is compiled by an AOT compiler. Unlike V8 and JSC, Typed JS is specifically designed for running Web applications on mobile devices with goals of having smaller memory footprint and binary size while achieving high-performance when integrated to mobile web applications. Unlike asm.js, which supports only a small set of operators, Typed JS supports most of JavaScript core design such as object model, prototype, functions and closures, and garbage collection. In brief, high performance is achieved by having fixed object layout, which allows us to access objects by using memory offsets. Also, by supporting AOT compilation, JavaScript VM is not required, and thus, the overhead from executing a VM is removed. This also results in having smaller memory footprint and binary size, as typical JavaScript virtual machine is replaced by a much compact native runtime library.

Other than the performance and smaller memory footprint, Typed JS provides additional advantages. Rigorous type checking in Typed JS improves productivity especially when implementing a large application, by early detecting errors and bugs that are caused by type-mismatching in the development phase. Typed JS is also portable such that, applications written in Typed JS runs on any platform without modifying the source code. However, it may need to be recompiled for each target platform. Furthermore, by distributing only binary files, application source code can be effectively hidden to prevent from unauthorized modification, which is often an important requirement in industry.

Finally, Typed JS can become the main language to easily implement efficient mobile applications. A binding mechanism between Typed JS and EFL[1] graphics library on Tizen mobile platform is implemented, and we successfully reimplemented mobile demo applications that came with Tizen SDK 2.3 originally written in C++ in Typed JS.

Organization: Section 2 presents related literature in the area of script languages. Section 3 presents design principles of Typed JS. Section 4 gives design, model, and implementation details of Typed JS. Section 5 shows experiment results. Lastly, we conclude in Sect. 6.

2 Related Work

JIT Compilers: Self [6] and StrongTalk [5] define many core techniques in JIT compilers such as polymorphic inline caches [11] and deoptimization [12]. Recent works on JIT are as follows. Bohm et al. [3] propose generalized trace JIT compilation approach that consider not only the paths through holes, but all frequently executed paths in a program. Rompf et al. [16] allows programs to invoke JIT compilation explicitly, as well as the JIT compiler to call back into the program to perform compile-time computation. In industry, Google's V8 [9] and Apple's JSC [2] are industry-leading VM and JIT compilers for JavaScript. PyPy [4] is a trace JIT framework written in Python.

[1] https://www.enlightenment.org/.

JavaScript with Types. Recently, a few techniques on extending JavaScript with types for better performance are proposed. TypeScript [14] is a superset of JavaScript that extends JavaScript with explicit types. Flow [8] is a subset of JavaScript that adds type-checks. SJS [7] shares the same concept as Flow, but it generates C++ code and supports AOT compilation. SJS integrates Wala [13] to perform type inference. Ahn et al. [1] propose how to derive an object type from its inherited prototype and method binding.

JavaScript VM and a Browser. A few techniques are proposed to run native code inside a browser. Doppio [17] is a JavaScript-based runtime system that allows C++ code and JVM programs run inside a browser. asm.js [15] converts C++ code to JavaScript to run on any JavaScript VM. However, it is not designed to support objects and prototype. Nacl (and PNaCl) [10] runs natively-compiled C++ code in a Chrome web browser.

3 Design of Typed JS

Typed JS utilizes type annotations, offers the usage of dynamic and static features, and an AOT compilation to increase the performance and yet reducing memory footprint on mobile platforms than current industry-leading JavaScript engines. The design principles of Typed JS are as follows.

- **Type Annotation**: Dynamicity of objects is one of major performance bottlenecks of JavaScript, as they require type-checked during runtime. Typed JS enforces types of objects to be specified when they are declared. It extends JavaScript syntax to accept type-annotated objects, and do not allow such objects to change its type during runtime. Figure 1 shows an example of a function written in Typed JS. In this example, the function is type-decorated.
- **Object Model**: Dynamically adding/deleting a property causes another performance decrease during runtime. To support dynamic objects, current JavaScript engines must check for the existence of a property during runtime. In addition to traditional dynamic objects in Typed JS, Typed JS supports sealed classes, which prevents users to change properties declared in sealed classes during runtime. Hence, sealed classes can be used if one prefers runtime efficiency over object dynamicity.
- **AOT Compilation**: Typed JS is compiled to a target-specific, optimized binary executable. Moreover, Typed JS can utilize modern compiler optimization techniques, as it annotates types and supports static classes. In our prototype, Typed JS compiler transpiles Typed JS source code into C++11, and it is natively compiled with g++'s optimization techniques.
- **Robust and Secure**: Typed JS follows the strict mode of JavaScript, and redefines a set of dynamic features that can be efficiently implemented. Also, Typed JS does not support evaluating source code during runtime, i.e., *eval*(), eliminating security holes.

```
var hanoi =
  function(disc: int, src: string,
           aux: string, dst: string): void {
    if(disc > 0) {
      hanoi(disc-1, src, dst, aux);
      console.log("Move disc " + disc +
                  " from " + src + " to " + dst);
      hanoi(disc-1, aux, src, dst);
  }
}
hanoi(5, "src", "aux", "dst");
```

Fig. 1. Tower of Hanoi in Typed JS

4 Architecture of Typed JS

Figure 2 shows the architecture of Typed JS. It consists of two major parts—compiler and runtime. The compiler part takes Typed JS source code as input, and generates C++11 code that heavily depends on internal data structure and Typed JS runtime library. The runtime part provides the implementation of the internal data structure and runtime library. The auto-generated C++11 code is compiled, and executed natively on a low-end Samsung mobile phone. This phone operates with Tizen[2] OS. In our implementation, C++ code is compiled with g++ found in Tizen-SDK-2.3.[3] This version of g++ generates binary for ARM architecture. Additionally, binding API, a selection of Tizen EFL graphics library wrappers, is also implemented to allow GUI components to be integrated with Typed JS. Each part further consists of smaller components, and they are explained in the following sections.

4.1 Compiler

Compilation is a 3-phase process, and each phase is explained as follows.

Parser. First, the parser generates an abstract syntax tree (AST) from Typed JS source code. The AST is extended from Mozilla JavaScript AST[4] to represent type-specific information as well as Typed JS extensions such as type-annotated objects and sealed classes. We also extend esprima,[5] an open source ECMAScript 5.1 parser to validate Typed JS syntax and generate an AST with Typed JS extensions. Figure 3 shows an AST of `hanoi()` from Fig. 1. The figure shows that new entries are added to type-annotate function parameters and return

[2] https://www.tizen.org.
[3] https://developer.tizen.org/.
[4] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API.
[5] http://esprima.org/.

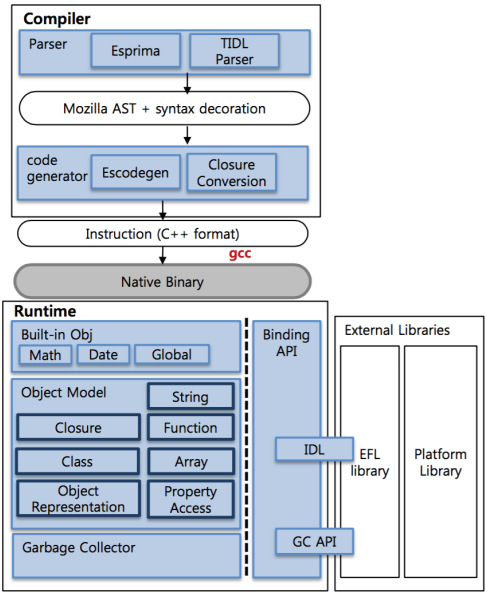


Fig. 2. Architecture of Typed JS

types (indicated by <---). Additionally, the Tizen EFL API written in Tizen IDL (TIDL) is parsed and validated, and corresponding API calls are also added to the AST (omitted in Fig. 3 due to complexity).

Code Generator. Second, the code generator takes an AST as input, performs semantics validation, and generates C++11 code. We modified escodegen,[6] which generates JavaScript code from an Mozilla JavaScript AST, to generate C++11 code with Typed JS runtime library. Furthermore, we modified escodegen to perform type inference by deriving and applying a set of type inference rules during the semantics validation to deduce unknown variable types before reporting a missing/invalid type error. Our type inference algorithm is a simplified version of Hindley-Milner.

The overview of type inference algorithm in Typed JS is given as follows. Given an expression of unknown type, its type is determined by deriving the type of right operand first, and assigning a compatible type (or subtype) to the left operand. Then, the type of the expression becomes the type of its left operand. This type resolution step is recursively applied from bottom up by traversing the AST until root is reached. For example, given an expression, $var\ x = 1$, the type of right operand is derived, which is a number by definition, and the same type as 1 is assigned to x. For an expression, $var\ y = x$, similar resolution process is performed once the type of its right operand, x, has been determined. The same process is applied to all different kinds of expressions in Typed JS

[6] https://github.com/estools/escodegen.

```
...
  "body": [{
    "type": "VariableDeclaration",
    "declarations": [{
      "type": "VariableDeclarator",
      "id": {
        "type": "Identifier",
        "name": "hanoi"
      },
    "init": {
      "type": "FunctionExpression",
      "params": [{
        "type": "VariableDeclarator",
        "id": {
          "type": "Identifier",
          "name": "disc"
        },
        "idType": {        <---
          "type": "Type",
          "name": "int"
        },
      },
    ...
    "returnType": {        <---
      "type": "Type",
      "name": "void"
    },
...
```

Fig. 3. An AST of hanoi()

whose types are not explicitly given by users. If the Typed JS function shown in Fig. 1 were written without explicitly declaring function parameter types, correct types would still be determined by the type inference rules. The type inference rules are still in preliminary phase, and further extension is left as future work.

Figure 4 shows a snippet of auto-generated C++ code of `hanoi()` shown in Fig. 1. The code shows that it utilizes Typed JS runtime library such as JSValue and JSStringRef to represent JavaScript number and string, respectively, while keeping the overall control flow intact (such as recursion and if-statement). A description of runtime model is provided in detail in Sect. 4.2.

Compilation. Finally, the auto-generated C++ code is compiled with g++ to produce an object binary (i.e., *.o). This binary is linked with Typed JS runtime library to produce an executable binary. Furthermore, when Tizen EFL bindings are required, the EFL wrapper API is also compiled and linked. Note that, the auto-generated C++ code is not necessarily optimal, but we implicitly apply compiler optimization (e.g., -O3) by using a state-of-the-art AOT compiler such as g++.

```
...
JSValue hanoi;
hanoi = JSValue([&](JSValue This, JSValue ___disc,
                    JSValue ___src, JSValue ___aux,
                    JSValue ___dst) mutable -> JSValue {
    int disc = (___disc).asInt32();
    JSStringRef src = (___src).asStringRef();
    JSStringRef aux = (___aux).asStringRef();
    JSStringRef dst = (___dst).asStringRef(); {
    if (disc > 0) {
        hanoi(JSValue(disc - 1), (src).asJSValue(),
              (dst).asJSValue(), (aux).asJSValue());
        console::log((JSStringRef("Move disc ") + disc +
                      JSStringRef(" from ") + src +
                      JSStringRef(" to ") + dst).asJSValue());
        hanoi(JSValue(disc - 1), (aux).asJSValue(),
              (src).asJSValue(), (dst).asJSValue());
    }
}   return undefined;
});
hanoi(JSValue(5), (JSStringRef("src")).asJSValue(),
      (JSStringRef("aux")).asJSValue(),
      (JSStringRef("dst")).asJSValue());
console::log((JSStringRef("success")).asJSValue());
...
```

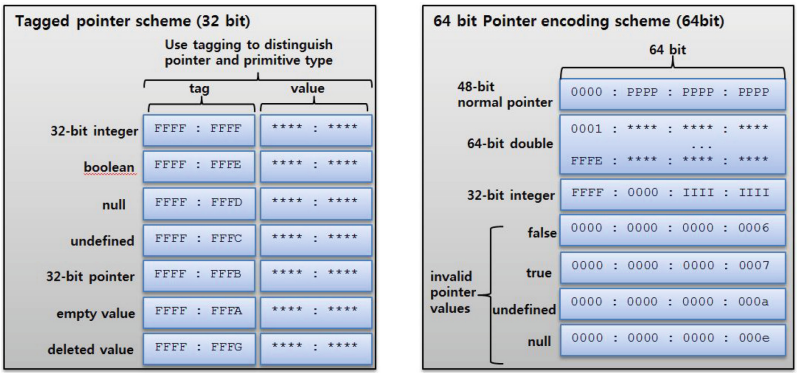
Fig. 4. hanoi() in C++11 (auto-generated)

4.2 Runtime

Runtime further consists of four components, and each component is explained below.

Object Model. In Typed JS, two object models are cohesively existed—dynamic object and sealed class models. Dynamic object model is the typical prototypical model found in ECMAScript 5.1, while sealed class model is the class-oriented model found in C++. Former is to be compatible with ECMASript 5.1 specification, while latter is designed to give better performance. The difference between dynamic and sealed class models is that, dynamically adding/deleting properties is removed in the sealed class model, and all property types are finalized in the compilation time. A sealed class is specifically designed to directly map a JavaScript object to a C++ class to gain further performance.

Typical JavaScript objects cannot be simply transpiled to C++ objects due to not supporting prototypical models in C++. To support dynamic objects, additional data structures are required, and these are explained as follows.



(a) 32-bit (b) 64-bit

Fig. 5. JSValue

Object Representation. Figure 5 shows a *JSValue* data structure used to represent an object in Typed JS. The tagged pointer scheme that JSValue uses allows either an object (regardless of its type) or a primitive value to be stored in a 64-bit block. It also allows itself to convert one type to another dynamically in runtime. For example, a variable of type boolean can be converted to a pointer type by updating the tag and value. JSValue is implemented differently depending on target machines. For 32-bit machines, it uses two 32-bit spaces. The first 32-bits are used to store a tag of an object, and the last 32-bits are used to store an actual value (either a primitive value or a pointer). An exception is when double is stored. On 64-bit machines, tagging is not explicitly used but similar approach applies. The physical size of JSValue is 64-bit for both 32 and 64-bit machines.

JS Object, Prototype and Sealed Classes. JS objects are connected to each other to represent a prototype chain in Typed JS. Figure 6 shows an example of JS objects, and how they are connected to represent a prototype chain.

An JS object is represented by a combination of JSValue and JSxImpl, where x is either Object, SealedClass, Function, String or Array, if the object is neither a number, null nor undefined. As discussed above, JSValue is a generic data container from which its value is used to retrieve the actual data that JSValue represents. For example, the JSValue for "tom" object in Fig. 6 points to an JSObjectImpl where properties of "tom" are implemented. Similarly, the JSValue for function *eat*() points to an JSFunctionImpl. For security reasons, JSxImpl objects cannot be directly accessed from Typed JS.

An JSObjectImpl implements a typical $(key, value)$ map to allow property reads and writes. The *key* and *value* is of type JSValue. Hence, a JSValue of any type can be stored as a property. For example, in Fig. 6, "tom" object contains three properties of type string, number and function, and they are all encapsulated in JSValues. Every JSObjectImpl contains a special *__proto__* as a key

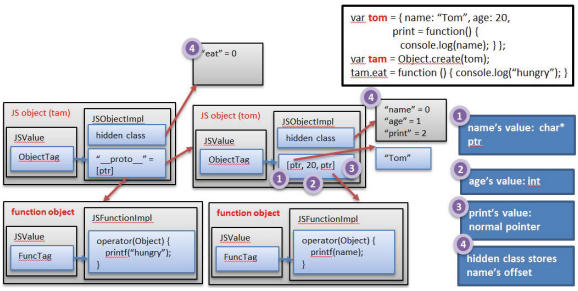


Fig. 6. Prototype chain in Typed JS

that points to a parent prototype object. By following *__proto__*, a prototypical model can be implemented. A property lookup of an object o can be performed as follows. First, a property x is searched in the (*key*, *value*) map in o. If a property x is found, its value is returned. Otherwise, the parent object y as indicated by *__proto__* is retrieved, and the same process is recursively applied. When *__proto__* is null, it indicates that, there is no parent prototype object, so the process terminates. Furthermore, to optimize property lookup, map is implemented using hidden class and inline caching techniques. Further description is given in Sect. 4.3.

An JSSealedClassImpl represents non-dynamic object. It contains a pointer that points to a typical C++ object. A sealed class is generated when a `class` is defined in Typed JS. For sealed classes, typical class-based inheritance is supported.

Function, Closure, and Polymorphic Functions. A function object (i.e., closure) is implemented by using the lambda function introduced in C++11. The lambda function allows us to define an anonymous function object that can be invoked and passed as an argument to a function. Given a Typed JS function, our code generator generates a lambda function accordingly. Polymorphic functions are supported by combining JSValue and lambda functions. A polymorphic function takes JSValues as input parameters, and the operation to perform on JSValue is determined by the type of the JSValue, which is determined in runtime. Figure 4 also shows how a function is represented by a C++11 lambda function with JSValues as input parameters. For the functions in sealed classes, it is not required to use JSValues, as types are not dynamic.

Supporting Modules. Runtime further consists of a number of supporting modules, and they are as follows.

– **Built-in Object**: Typed JS provides a number of built-in objects specified in ECMAScript 5. Some examples include Math, Date, Global, etc. These built-in objects can be used from both dynamic and static objects.

- **Garbage Collector**: In Typed JS, automatic memory management is provided when objects are allocated or deleted. In our prototype, we adopted Boehm-Demers-Weiser conservative garbage collector,[7] when objects are allocated and removed. For future work, we plan to implement reference counting-based memory management, as it gives less overhead on mobile platforms.
- **A Graphics Binding API**: The graphics binding API gives us many opportunities to use Typed JS on mobile devices. In our prototype, the binding API wraps a number of EFL graphics library on Tizen. This allows us to access Tizen's graphics components, and we successfully reimplemented mobile demo applications that came with Tizen SDK 2.3 originally written in C++ in Typed JS. This provides a way of writing native Tizen application in JavaScript-like style. A possible future work is to let the binding API wrap a web browser's canvas API to run web applications natively on Tizen.

4.3 Optimization

Noticeable optimization techniques that Typed JS currently implements are as follows.

- **Fast Property Access**: For fast property access, we implemented hidden class and inline caching technique used in V8. This technique caches a property offset when a property in an object is accessed frequently, and when the same property is accessed after caching, it improves the property access performance by quickly reading the value located in the offset without performing any string operation, as long as the layout of the object that contains the property is not changed. This allows us not to perform string hashing every time when a property is accessed.
- **Fast String Operation**: Poor string implementation significantly affects the overall performance of Type JS. Currently, we implemented string interning and rope string to improve basic string operations.

4.4 Limitations

Current implementation of Typed JS poses the following limitations.

- **Debugging Symbols**: Since Typed JS transpiles JavaScript code into C++ code and compiles using g++, generating and maintaining debugging symbols and supporting a debugger becomes a challenging problem. To address this problem, in future work, we are planning to generate LLVM[8] IR instead of C++ code from Typed JS. In this way, we can maintain a debugging symbol table and add debugging information between LLVM IRs to allow us to use any LLVM-supported debugger.

[7] http://www.hboehm.info/gc.
[8] http://llvm.org.

- **Supporting Existing JavaScript Libraries**: While Typed JS supports most of JavaScript operations, it is often insufficient to support existing JavaScript libraries as-is such as the libraries found in node.js.[9] This is because many libraries use unsupported patterns such as *eval*(), which requires the code to be rewritten without using such patterns. To address this problem, in future work, we are planning to generate a Typed JS pattern analyzer that can detect unsupported patterns in existing JavaScript code and even suggests valid code for Typed JS.
- **Security**: The current approach of generating and compiling C++ code from Typed JS can pose a security risk, as the auto-generated code can be altered before it is compiled. Using LLVM instead of g++ can solve this problem, as LLVM IR is not written to disk when it is compiled.

5 Experimental Results

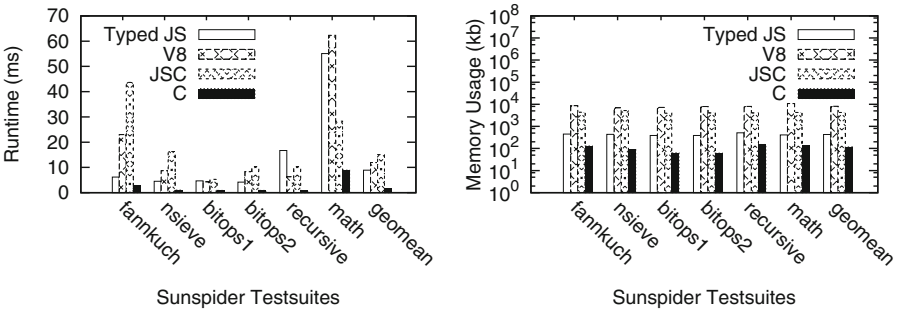
We now present experimental results. Experiments were conducted on a preproduction, low-end Samsung mobile phone running Tizen 2.3. Tizen is an open source mobile operating system currently developed by Samsung Software R&D Center. In addition, to compare the performance and memory usage of Typed JS in an unlimited-resource environment, the same set of experiments were repeated on a 3.5 GHz Linux desktop with 16 GiB of memory. Typed JS is compiled using Tizen SDK 2.3 and executed on the Samsung mobile phone. For Linux desktop, g++ was used to compile Typed JS.

For performance measurement, Sunspider JavaScript Benchmark[10] was used. Memory usage was measured by checking the RSS size used by each test suite process. Binary size was measured by adding all shared libraries required to run each test suite. To compare the performance and memory usage of Typed JS against existing JavaScript engines, the same test suites were executed on V8 and JSC. When measuring the performance of V8 and JSC, the initial loading time for JavaScript virtual machine were excluded. Furthermore, the same test suites were also ported to C and executed. The ported C test suites were compiled with -O3 optimization. In the experiments, this C implementation defines the practical upper bound for both performance and memory usage.

Figure 7 shows the runtime performance and memory usage of Typed JS against V8, JSC, and C on the Tizen mobile phone (Note the log scale in Fig. 7(b)). Typed JS outperforms V8 and JSC by up to 3.5x while consuming up to 20x less memory. Smaller memory usage is largely due to not using a virtual machine unlike others. Typed JS is outperformed by JSC (and V8) on `recursive` and `math` test suites due to inefficient lambda functions in C++11 and lack of JIT compilation, respectively.

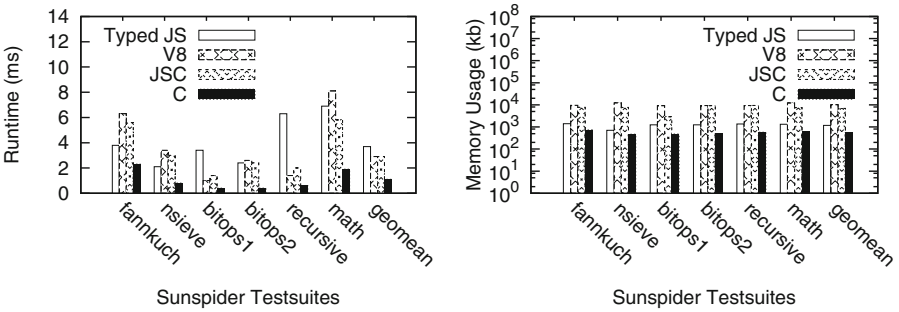
Figure 8 repeats the same experiments on the Linux desktop. Similar results are observed, but the performance gap between Typed JS and other engines are much reduced due to more powerful CPU. Furthermore, we observe that,

[9] http://nodejs.org.
[10] http://www.webkit.org/perf/sunspider/sunspider.html.



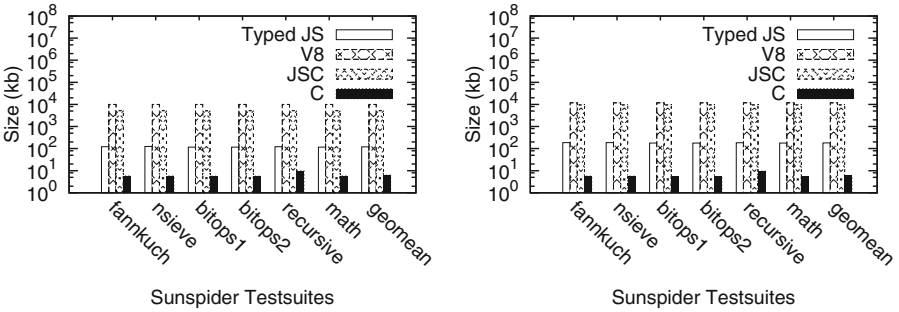
(a) Runtime (b) Memory

Fig. 7. Tizen phone



(a) Runtime (b) Memory

Fig. 8. Linux desktop



(a) Tizen phone (b) Linux desktop

Fig. 9. Binary size

in general, the same test suite runs up to 10x slower on a mobile phone than desktop.

Figure 9 shows the binary size of Typed JS against V8, JSC, and C. The binary size of Typed JS is orders of magnitude smaller than that of V8 and JSC. Smaller binary size is also related to not using a virtual machine and utilizing C library, which is generally more compact than JavaScript library.

6 Conclusion

In this paper, we have presented Typed JS, a memory efficient but yet high-performance JavaScript engine for mobile devices. By utilizing type-decoration, Typed JS can be compiled ahead-of-time, which results in achieving smaller memory footprint and high-performance than traditional virtual machine-based JavaScript engines without scarifying much of core JavaScript concepts, such as objects, prototype, etc. Experiments show that Typed JS is memory-efficient and achieves better performance compared to industry-leading JavaScript engines on Tizen mobile platform.

There are several possibilities for future work. First, we plan to update C++11 code generator to generate LLVM IR to enhance performance, usability, and security. Second, implementing more rigorous type inference rules is planned. Third, power consumption evaluation and performance measurements on other mobile platforms are planned. Lastly, reference counting-based memory management is also planned.

Acknowledgment. We thank our group members Junyoung Cho, Eunji Jeong, Saebom Kim, Wonyong Kim, Sanggyu Lee, Seungsoo Lee, Jaeman Park, and Youngsoo Son for their contributions to this paper. We are also grateful to the anonymous reviewers for their constructive comments on this paper.

References

1. Ahn, W., Choi, J., Shull, T., Garzarán, M.J., Torrellas, J.: Improving javascript performance by deconstructing the type system. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 51. ACM, Edinburgh, United Kingdom, June 2014
2. Apple.Javascriptcore (2005). http://trac.webkit.org/wiki/JavaScriptCore
3. Böhm, I., von Koch, T.J.K.E., Kyle, S.C., Franke, B., Topham, N.P.: Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 74–85. ACM, San Jose, CA, June 2011
4. Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: Pypy's tracing JIT compiler. In: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, pp. 18–25. ACM, Genova, Italy (2009)

5. Bracha, G., Griswold, D.: Strongtalk: typechecking smalltalk in a production environment. In: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 215–230. ACM, Washington, DC, October 1993
6. Chambers, C., Ungar, D., Lee, E.: An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. Lisp Symb. Comput. **4**(3), 243–281 (1991)
7. Choi, P.W., Chandra, S., Necula, G., Sen, K.: SJS: a typed subset of javascript with fixed object layout. Technical report UCB/EECS-2015-10, EECS Department, University of California, Berkeley, March 2015
8. Facebook.flow (2014). http://flowtype.org/
9. Google.Chrome v8 (2008). https://developers.google.com/v8/
10. Google.NaCl and PNaCl (2013). https://developer.chrome.com/native-client/nacl-and-pnacl/
11. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: Proceedings of European Conference on Object-Oriented Programming, pp. 21–38. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Geneva, Switzerland, July 1991
12. Hölzle, U., Chambers, C., Ungar, D.: Debugging optimized code with dynamic deoptimization. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 32–43. ACM, San Francisco CA, June 1992
13. IBM.Wala (2006). https://wala.sourceforge.net
14. Microsoft.TypeScript (2012). http://www.typescriptlang.org/
15. Mozilla.asm.js (2013). http://asmjs.org/
16. Rompf, T., Sujeeth, A.K., Brown, K.J., Lee, H., Chafi, H., Olukotun, K.: Surgical precision JIT compilers. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 8. ACM, Edinburgh, United Kingdom, June 2014
17. Vilk, J., Berger, E.D.: Doppio: breaking the browser language barrier. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 52. ACM, Edinburgh, United Kingdom, June 2014