Static TypeScript

An Implementation of a Static Compiler for the TypeScript Language

Thomas Ball
Microsoft Research
Redmond, WA, United States
tball@microsoft.com

Peli de Halleux
Microsoft Research
Redmond, WA, United States
jhalleux@microsoft.com

Michał Moskal
Microsoft Research
Redmond, WA, United States
mimoskal@microsoft.com

Abstract

While the programming of microcontroller-based embeddable devices typically is the realm of the C language, such devices are now finding their way into the classroom for CS education, even at the level of middle school. As a result, the use of scripting languages (such as JavaScript and Python) for microcontrollers is on the rise.

We present Static TypeScript (STS), a subset of TypeScript (itself, a gradually typed superset of JavaScript), and its compiler/linker toolchain, which is implemented fully in TypeScript and runs in the web browser. STS is designed to be useful in practice (especially in education), while being amenable to static compilation targeting small devices. A user's STS program is compiled to machine code in the browser and linked against a precompiled C++ runtime, producing an executable that is more efficient than the prevalent embedded interpreter approach, extending battery life and making it possible to run on devices with as little as 16 kB of RAM (such as the BBC micro:bit).

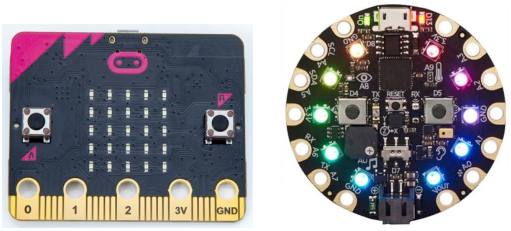
This paper is primarily a description of the STS system and the technical challenges of implementing embedded programming platforms in the classroom.

Keywords JavaScript, TypeScript, compiler, interpreter, microcontrollers, virtual machine

1 Introduction

Recently, physical computing has been making headway in the classroom, engaging children to build simple interactive embedded systems. For example, Figure 1(a) shows the BBC micro:bit [1], a small programmable Arduino-inspired computer with an integrated 5x5 LED display, several sensors and Bluetooth Low Energy (BLE) radio technology. The device first rolled out in 2015 to all year 7 students (age 10 to 11) in the UK and has since gone global, with four million units distributed worldwide to date via the micro:bit Education Foundation (https://microbit.org). Figure 1(b) shows a different educational device featuring RGB LEDs: Adafruit's Circuit Playground Express (CPX).

Research suggests that using such devices in computer science education increases engagement, especially among girls, increases confidence in both students and teachers, and makes lessons more fun [2, 16].



(a) (b)

Figure 1. Two Cortex-M0 microcontroller-based educational devices: (a) the BBC micro:bit has a Nordic nRF51822 MCU with 16 kB RAM and 256 kB flash; (b) Adafruit's Circuit Playground Express (https://adafruit.com/products/3333) has an Atmel SAMD21 MCU with 32 kB RAM and 256 kB flash.

To keep costs low for schools, these devices typically employ 32 bit ARM Cortex-M microcontrollers (MCUs) with 16-256kB of RAM and are programmed using an external computer (usually a laptop or desktop). Programming such devices in a classroom presents a number of technical challenges:

(1) the selection/design of an age-appropriate programming language and environment;
(2) classroom computers running outdated operating systems, having intermittent and slow internet connectivity, and locked down by school IT administrators, which makes native app installation difficult;
(3) the transfer of the student's program from the computer to the device, where it can run on battery power (as many projects embed the device in an experiment or "make").

With respect to these challenges, there are various embedded interpreters for popular scripting languages, such as JavaScript (JerryScript [8, 15], Duktape [22], Espruino [23], mJS [20], and MuJS [19]) and Python (MicroPython [9] and its fork CircuitPython [12]). The interpreters run directly on the MCU, requiring just the transfer of program text from the host computer, but forego the benefits of advanced optimizing JIT compilers (such as V8) that require about two orders of magnitude more memory than is available on MCUs.

MPLR 2019, Under submission,
.

Figure 2. Three microcontroller-based game handhelds with 160x120 color screens. These boards use ARM's Cortex-M4F core: the ATSAMD51G19 (192kB RAM, running at 120Mhz) and STM32F401RE (96kB RAM, running at 84Mhz).

Unfortunately, such embedded interpreters are between one and three orders of magnitude slower than V8 (see Section 4), affecting responsiveness and battery life. Even more importantly, due to the representation of objects in memory as dynamic key-value mappings, the memory footprint can be several times that of an equivalent C program. This can severely limit the applications that can be deployed on low-memory devices such as the micro:bit (16 kB RAM) and CPX (32 kB RAM).

1.1 Static TypeScript

As an alternative to embedded interpreters, we present Static TypeScript (STS), a syntactic subset of TypeScript,[3] supported by a compiler (written in TypeScript) that generates machine code that runs efficiently on MCUs in the target RAM range of 16-256kB. The design of STS and its compiler and supporting runtime were dictated primarily by the above three challenges. In particular:

- STS eliminates most of the "bad parts" of JavaScript; following StrongScript [14], STS uses nominal typing for statically declared classes and supports efficient compilation of classes using classic techniques for v-tables.
- the STS toolchain runs offline, once loaded into a web browser, without the need for a C/C++ compiler – the toolchain, implemented in TypeScript, compiles STS to Thumb machine code and links this code against a pre-compiled C++ runtime in the browser, which is often the only available execution environment in schools.
- the STS compiler generates surprisingly efficient and compact machine code, which unlocks a range of application domains such as game programming for low-resource devices such as those in Figure 2, all of which were enabled by STS.

Deployment of STS user programs to embedded devices does not require app or device driver installation, just access to a web browser. Compiled programs appear as downloads, which are then transferred manually by the user to the device, which appears as a USB mass storage device, via file copy (or directly through WebUSB, an upcoming standard for connecting websites to physical devices).

The relatively simple compilation scheme for STS (presented in Section 3) leads to surprisingly good performance on a collection of small JavaScript benchmarks, often comparable to advanced, state of the art JIT compilers like V8, with orders of magnitude smaller memory requirements (see Section 4). It is also at least an order of magnitude faster than the embedded interpreted approach. A novel aspect of evaluation is a comparison of different strategies for dealing with field/method lookup spanning classes, interfaces, and dynamic maps.

1.2 MakeCode: Easy Embedded for Education

STS is the core language supported by the MakeCode Framework.[1] MakeCode enables creation of custom programming experiences for MCU-based devices. Each MakeCode experience (we often call them editors, though they also bundle a simulator, APIs, tutorials, documentation, etc.) targets programming of a specific device or device class via STS. [7]

Most MakeCode editors are deployed primarily as web apps, including a full-featured text editor for developing STS programs based on Monaco (the editor component of Visual Studio Code), as well as a graphical programming interface based on Google's Blockly framework (STS metadata in comments defines the mapping from STS APIs to Blockly and MakeCode translates between Blockly and STS).

The MakeCode editors, including the primary coding experiences for BBC micro:bit and for Adafruit CPX,[2] have been used by millions of students and teachers worldwide to date.

STS supports the concept of a package, a collection of STS, C++ and assembly files, that also can list other packages as dependencies. This capability has been used by third parties to extend the MakeCode editors, mainly to accommodate hardware peripherals for various boards.[3] Notably, most of the packages avoid pitfalls of unsafe C/C++ completely and are authored solely in STS, due to the efficiency of the STS compiler and the availability of low-level STS APIs for

[1] See https://makecode.com. The framework, along with many editors, is open source under MIT license, see https://github.com/microsoft/pxt.
[2] See https://makecode.microbit.org and https://makecode.adafruit.com.
[3] For example for micro:bit, see https://makecode.microbit.org/extensions

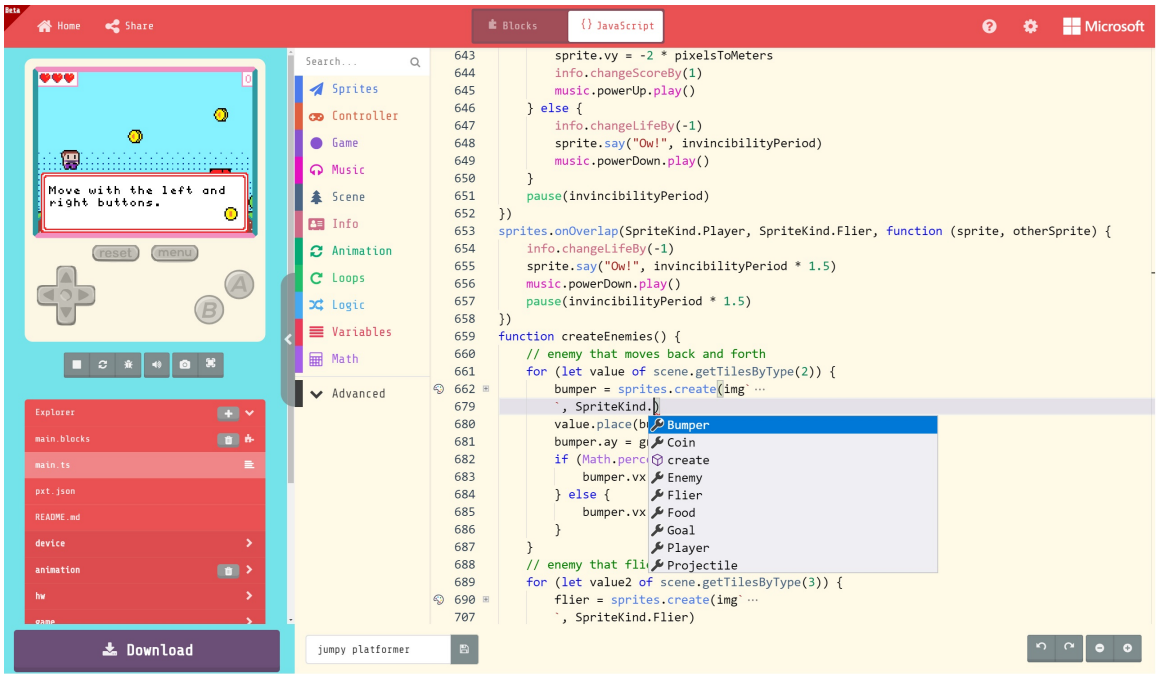


Figure 3. MakeCode Arcade editor. The left pane is the simulator for the arcade device; the middle pane is the categories of APIs available in the editor; the right pane is the Monaco editor with STS user code for a platformer game (https://makecode.com/85409-23773-98992-33605).

The toggle on top is used switch between Blocks and Static TypeScript (labelled JavaScript for marketing reasons).

accessing hardware via digital/analog pins (GPIO, PWM and servos) and serial protocols (I2C and SPI).

Figure 3 shows the MakeCode Arcade editor created for programming the handheld gaming devices from Figure 2 (in fact, the STS program shown in the editor is the one deployed to the three devices - it is a simple platformer game). MakeCode Arcade includes a game engine written almost completely in STS, and thus places high requirements on code efficiency to achieve pleasing visual effects at high frame rates. The game engine includes the game loop, stack of event contexts, physics engine, text and line drawing, as well as game-specific frameworks (eg., for platformer games), in all about 10,000 lines of STS code, with only the most basic image-blitting primitives implemented in C++. The games built with Arcade run either in the web browser (on desktop or mobile), or on various models of dedicated hardware featuring a 160x120 pixel 16 color screen and an MCU running at around 100MHz with around 100kB of RAM.

The primary contribution of this paper is a description of a widely deployed system and ways in which it addresses the classroom-specific problems listed above.

2 Static TypeScript (STS)

TypeScript [3] is a gradually-typed [18] superset of the JavaScript language. This means that every JavaScript program is a TypeScript program and that types can be optionally added, as needed, which enables better IDE support and error checking for larger JavaScript programs. By design, TypeScript provides no type soundness guarantees. Object types provide a unification of maps, functions, and classes; structural subtyping between object types defines substitutability and compatibility checks. Type erasure (and minor syntactic transformations) yields a raw JavaScript program.

STS is a subset of TypeScript that excludes many of the highly dynamic parts of JavaScript: the `with` statement, the `eval` expression, prototype-based inheritance, the `this` pointer outside classes, the `arguments` keyword, and the `apply` method. STS retains features such as dynamic maps, but keeps them separate from the nominal class abstraction. Such restrictions are acceptable because most beginners' programs are quite simple, and there are very few existing JavaScript or TypeScript libraries in the embedded space, so the chance of these using JavaScript features such as monkey patching is small.

Our goal is not to grow STS to the point of supporting all of TypeScript. Instead, we follow the pragmatic approach of adding features useful in embedded context, as guided by user feedback.

In contrast to TypeScript, where all object types are bags of properties, STS has at runtime four kinds of unrelated object types:

1. a *dynamic map* type has named (string-indexed) properties that can hold values of any type;
2. a *function* (closure) type;
3. a *class* type describes instances of a class, which are treated nominally, via an efficient runtime subtype check on each field/method access, as described below;
4. an *array* (collection) type.

In this sense, STS is much closer to spirit to Java and C# in their treatment of types as "protectors of abstractions", unlike JavaScript which allows a much more freeform treatment of an object's role. As discussed in Section 3.4, runtime type tags are used to distinguish the different kinds of built-in object types listed above (as well as primitives like boxed numbers and strings).

As in TypeScript, type casts do not generate any code, and thus always succeed. Instead, STS protects the nominal class abstraction at the point of field/method access. Just as `x.f` causes a runtime error in JavaScript when `x == null`, executing `(x as T).f` will cause a runtime error in STS if `T` is a class with field `f`, and the dynamic type of `x` is not a nominal subtype of `T`. If `T` is an interface, `any`, or some complex type (eg., union or intersection), then the field will be looked up by name regardless of dynamic type of `x`.

Other abstractions are also protected at runtime, as in JavaScript: for example, making a function call on something that is not of *function* type. Currently, it is an error to dynamically add a new property to any type other than the *map* type. Restrictions to the dynamic JavaScript semantics may be lifted in the future, depending on user feedback. To date, these restrictions have generated no concerns among our user community (both educators and developers).

STS primitive types are treated according to JavaScript semantics. In particulars, all numbers are logically IEEE 64 bit floating point, but 31-bit signed tagged integers are used where possible for performance. Implementation of operators, like addition or comparison, branch on the dynamic types of values to follow JavaScript semantics, with the fast integer path hand-implemented in assembly.

The design goal of STS to be both syntactic and semantic subset of TypeScript, by which we mean that if a program compiles successfully in STS, it will have the same semantics as the TypeScript program, or it will crash (in the circumstances listed above).

2.1 C++ Interop

STS programs running on MCUs are supported by a runtime implemented in C++, C, and assembly. The runtime implements language primitives (operators, collections, support for classes, dynamic maps, etc.), as well as allows access to the underlying device hardware. The runtime can be extended by packages as explained in Section 2.2.

STS supports calling functions from C++ to STS and vice versa. To simplify this process, STS uses a simple code-generation scheme, where a special comment on a C++ function indicates that it should be exported to STS (`//% `), as well as to Blockly (`//% block`). A build step parses the C++ code in search of these comments, it also collects the prototypes (signatures) of these functions, so appropriate conversions can be generated when calling them. For example:

```
// source C++ code:
namespace control {
  /** Register an event handler */
  //% block
  void onEvent(int eventType, Action handler) {
    // arrange for pxt::runAction0(handler)
    // to be called when eventType is triggered
  }
}

// generated TypeScript:
declare namespace control {
  /** Register an event handler */
  //% block shim=control::onEvent
  function onEvent(eventType: number,
                   handler: () => void) : void;
}
```

The C++ namespaces and function names are mapped directly to their STS equivalents and the documentation comment is copied verbatim. The comment `//% block`, which indicates that the function should be exposed as a Blockly graphical block is also copied. There are many other possible comments controlling the look and feel of the graphical equivalent. The STS function also gets an additional `shim` annotation, which indicates the name of the corresponding C++ function (in some cases the STS declaration is written by hand, and the name of the C++ function does not have to match the STS function).

The C++ types are mapped to STS types. Since in STS all numbers are conceptually doubles[4], the C++ `int` is mapped to STS `number`. When the C++ function is called, the STS compiler makes sure to convert the value passed to an integer. Other C++ integer types (eg., `uint16_t`) are supported in a similar way. The C++ `Action` type represents a reference to a closure, which can be called with `pxt::runAction0()`.

While class methods are not supported directly, regular functions can be used to implement objects. For example:

```
// C++
typedef BoxedBuffer *Buffer;
namespace BufferMethods {
  //%
  int getByte(Buffer self, uint32_t i) {
    return i < self->len ? self->data[i] : 0;
```

[4]There is some support to store numbers as integers of various sizes to save memory, but it only applies to storage, not intermediate computations.

```
    }
}

// TypeScript
interface Buffer {
    //% shim=BufferMethods::getByte
    getByte(i: number);
}
```

All functions in `BufferMethods` namespace must take `Buffer` as first argument and are exposed as members of the `Buffer` class on the STS side. When such members are called, the STS compiler will make sure the first argument is non-null and a subtype of `Foo`. These interfaces are conceptually better understood as non-extensible classes with opaque representation, that is they cannot be implemented by regular classes, and member resolution is static. The interface syntax was chosen because TypeScript allows extending interfaces with new methods across files. We allow such additions, provided the new methods have the `shim=...` annotation described above, or alternatively an analog annotation specifying a TypeScript, not C++, replacement function. This usually only concerns authors of advanced C++ packages (see below).

2.2 Packages

STS supports multiple input files. It also supports TypeScript `namespace` syntax for scoping. Files do not introduce scopes and JavaScript modules are currently not supported. The input files come from one or more *packages*. There is one main package, which can list other packages as dependencies, which can in turn list further dependencies. There are multiple ways of specifying versions of packages, including built-in packages, file paths when operating from command line, and URLs of GitHub repositories. There can be only one version for each package (otherwise we would likely get redefinition errors).

MakeCode editor builders will generally decide to include a number of built-in packages, which ship with the editor. These can be further extended with packages coming from GitHub. The MakeCode web app has support for authoring and publishing packages to GitHub. Because namespaces are independent of files, it is easy for packages to extend existing namespaces. Currently, STS does not enforce any discipline here.

MakeCode comes with a number of packages that editor builders can include (common packages). They provide support for various hardware features (pins, buttons, buzzer, screen, etc.), as well as higher-level concepts like sprite-handling game library. Some of these packages come in variants, sharing interface but with different implementations (eg., drivers for different screens).

External (GitHub) packages typically provide support for other hardware peripherals. Users typically do not use too many external packages at once, so we feel the risk of name conflicts due to lack of namespace enforcement is low, and it allows for fitting new APIs naturally in existing namespaces.

3 Compiler and Runtime

The STS compiler and toolchain (linker, etc.) are written solely in TypeScript. There currently is no support for separate compilation of STS files: STS is a whole program compiler (with support for caching precompiled packages, which includes the C++ runtime). The STS device runtime is mainly written in C++ and includes a bespoke garbage collector. As mentioned before, it is not a goal to generalize STS to support full JavaScript.

3.1 Compiler Toolchain

The source TypeScript program is processed by the regular TypeScript compiler to perform syntactic and semantic analysis, including type checking. This produces type-annotated abstract syntax trees (ASTs) that are then checked for constructs outside of STS (`eval`, `arguments`, etc.). The AST is then transformed into a custom intermediate representation (IR) with language constructs desugared to calls to runtime functions. This IR is later transformed into one of three forms:

1. continuation passing JavaScript for execution within the browser (inside of a separate "simulator" IFrame)
2. ARM Thumb machine code, linked with pre-compiled C++ runtime and executed on bare-metal hardware or inside of an operating system
3. bytecode for a custom VM interpreter, meant for platforms where where dynamic code loading/generation is impossible (like XBox or iOS)

Both the ARM Thumb and the custom bytecode are generated in form of assembly code, and translated to machine code by a custom assembler. In this section we focus on the native 32-bit ARM Thumb target (though we compare the performance of the VM, see Section 4.2).

The regular TypeScript compiler, the STS code generators, assembler, and linker are all implemented in TypeScript and run both in the web browser and on command line.

3.2 Linking

The generated machine code is linked to a pre-compiled C++ runtime. C++ compilation takes place in a cloud service, with the resulting runtime cached both in the cloud content delivery network, and in the browser (caching is based on a strong hash of all the C++ sources, options etc.). Typically, the C++ runtime does not change while the user is working on their program, allowing for offline operation.[5]

[5] While it could be possible to compile the C++ code locally using Emscripten or similar technologies, the compilation toolchain, header files, and libraries would likely require tens of megabytes of download straining offline storage in the browser.

The generated machine code is generally appended at the end of the compiled runtime. Depending on the file format (in particular for ELF) of the target device, the resulting file needs to be patched slightly. To generate code, the assembler needs to know addresses of runtime functions. These are extracted from the runtime binary.

It is also possible for packages to include C++ code that extends the runtime. Each such combination of C++-containing packages have to compiled and cached separately. This is done transparently by the cloud service, and the potential exponential number of combinations have so far has not proven to be a problem in practice, as students do not use many external packages at once, and our experience has shown that few of the packages written for MakeCode utilize native C++.

3.3 Representation of values

The compiled programs use a regular C++ stack for local variables, return addresses, and temporary computation. Values stored on the stack and inside of objects generally use a uniform, type-independent representation, that is always 32-bits long.

The number n such that $-2^{30} \leq n < 2^{30}$ is represented by the 32-bit value $2n + 1$, so that the lowest-order bit is set. Other numbers, are represented as regular objects, specifically boxed 64 bit doubles.

Special constants, such as `true`, `false`, and `null` are represented by specific 32-bit values, such that the lowest-order bit is cleared, and the next to lowest-order is set (there is encoding space for 2^{30} such values, but only a few are currently used). The JavaScript `undefined` is represented by `0` (in other words, the C++ `NULL`), since it is the default value for memory allocation and also for JavaScript fields and variables.

All other values are represented as pointers to either constant flash-allocated objects or objects allocated on the heap. All pointers are word-aligned (32 bit), so the lower-order two bits are cleared. For objects created by the STS runtime the first word is occupied by a pointer to a virtual table (see Section 3.4).[6]

3.4 Virtual table layout

STS classes are compiled similarly to Java or C# with single inheritance and static memory layout. The first word of an object points to a virtual table, and subsequent words are allocated for fields (with fields of the base class, if any, coming first). The virtual table contains object size, a statically allocated class number (this includes the tags for the builtin types supported by STS, as well as user-defined types), a pointer to the interface table and interface hash (see below), followed by pointers to methods.

The first four method slots are pre-allocated for runtime and GC-related functions related to object marking and deallocation, which follow C++ calling conventions. The remaining functions follow STS calling convention. First of STS method pointers is `toString()` if defined[7]. Next there are methods from the base class if any, followed by methods of the current class. If a method is never called with dynamic dispatch (for example, because it is never overridden, or never called), it is not included in the table (for example, in the MakeCode Arcade game engine only 13% of methods use dynamic dispatch).

The interface table contains descriptors for all fields and methods of the class in order of definition. Each descriptor contains the member index (assigned globally to all member names in the program) and a function pointer. Field descriptors also contain the offset of the field within the class. The descriptors are used in member lookup and also when iterating over object properties (eg., using `Object.keys()`). The descriptors are indexed by a simple hash table, where the keys are computed by multiplying the member index by the interface hash multiplier (computed per-class at compile time) fetched from the virtual table.

STS employs three methods of member lookup:

- when the receiver is statically of a class type C, the compiler generates a runtime subtype check against C (which may fail, as the dynamic type may not be a subtype of C) and then uses a direct offset into either the virtual table for methods, or into the object itself for field accesses;
- otherwise, if the receiver is dynamically of a class type (statically it could be an interface, `any`, or some more complex structural type), the member descriptor is looked up in the interface table using the member index and the interface hash key;
- otherwise, it is a type-specific function for objects implemented in the C++ runtime, in particular the dynamic map used when compiling object literals or `new Object()`.

Section 4.3 compares the performance of the three methods.

3.5 Arithmetic operators

For some arithmetic operators, the fast integer path is written in assembly for speed. For example, here is the implementation of arithmetic +:

```
_numops_adds:
    ands r2, r0, r1   ; r2 := r0 & r1
    ands r2, #1       ; r2 &= 1
    beq .boxed        ; last bit clear?
    subs r2, r1, #1   ; r2 := r1 - 1
```

[6] An alternative representation, where numbers are represented as $2n$ and pointers as $p + 1$ would be problematic since ARM Thumb only has word-loading instructions with offsets that are multiples of 4.

[7] `toString()` and `valueOf()` play special role in JavaScript runtime conversion semantic. `valueOf()` is currently not supported.

```
    adds r2, r0, r2   ; r2 := r0 + r2
    bvs .boxed        ; overflow?
    mov r0, r2        ; r0 := r2
    bx lr             ; return
.boxed:
    mov r4, lr        ; save return address
    bl numops::adds   ; call into runtime
    bx r4             ; return
```

Other operators with specialized implementations are `-`, `|`, `&`, `^` and conversion to integer (used when calling C++ runtime functions). This specialized assembly gives about 2x speedup compared to always calling the C++ function, which we do for example for multiplication (see Section 4.3). Other operators are implemented in C++ as functions taking abstract values with branches for integers, boxed numbers, and other types.

3.6 Representation of built-in objects

Arrays are similar to C++ standard vectors, but with more conservative growing strategy. Sparse arrays are not supported. Simple array accesses are implemented directly in assembly, including range checking. Cases that involve conversions of indices or growing the array are handled by the C++ runtime.

Buffers are simply continuous chunks of memory, with assembly byte accessors, and a number of additional utility methods implemented in C++.

Strings come in four different representations, each with a v-table pointer at the beginning. All strings are currently limited to 65,535 bytes. ASCII strings (where all characters are in range 0-127) are represented using a length prefix followed by NUL-terminated character data (there can still be NUL characters inside, but the final NUL is added for convenience of C++ functions). Short Unicode strings are represented similarly, using UTF-8 and length expressed in bytes, but with a different v-table. The indexing methods decode the UTF-8 on the fly.

Longer Unicode strings are represented as the length of the string in characters, the size of the string in bytes, and a pointer to data, that contains the actual character data in UTF-8 (again, NUL-terminated), and a skip list that contains a byte offset for every character offset divisible by 16. Indexing methods start at closest preceding skip list offset and decode UTF-8 data from there. This encoding is used instead of standard UTF-16 to save space. It also ensures that all strings (except for temporary ones described below) contain a valid UTF-8 NUL-terminated string, making it easier for the C++ runtime functions to handle them without additional conversions.

Finally, cons-strings are allocated when two long strings are concatenated together. They consist of two pointers to strings (which themselves can be cons-strings). When a cons-string is indexed, it is transformed in-place into a skip-list string (they are both 12 bytes). This makes string concatenation, which is fairly common in JavaScript, a constant time operation. This is an established optimization technique [4], used in all major JavaScript engines.

The first three kind of strings are emitted by the compiler in flash, while all four can be constructed dynamically.

Closures are represented as a pointer to static code of the function followed by read-only locals captured from outer scopes (top-level globals are not included). If a variable can be written after capture, it is transformed (in all scopes) into a pointer to a heap object that holds its value. A specific register is statically allocated to hold the pointer to the closure object, during closure execution.

Functions that are used as values, but do not capture anything, are allocated statically in the above format in the flash.

Dynamic maps are simply two vectors, one of keys and one of values. Lookup is linear.

3.7 Peep-hole optimizations

After code generation, the resulting assembly is run through a simple peep-hole optimizer. The particular instructions sequences to be optimized were identified by a script run on large piece of generated code. The script would identify 2-3 instruction sequences and sort them by number of occurrence. The top ones were then inspected by hand to see if they could be simplified. Typical peep-hole rules are as follows:

```
push {lr}; push {X, ...} -> push {lr, X, ...}
pop {X, ...}; pop {pc}   -> pop {X, ..., pc}
push {rX}; pop {rX}      -> nothing
push {rX}; pop {rY}      -> mov rY, rX
pop {rX}; push {rX}      -> ldr rX, [sp, #0]
push {rX}; ldr rX, [sp, #0]      -> push {rX}
push {rX}; movs rY, #V; pop {rX} ->
     movs rY, #V (when X != Y)
```

For branches, the compiler always generates a fully general, but cumbersome instruction sequence, which is then simplified by the peep-hole optimizer, eg. when `X` is in short jump range:

```
beq .skip; b X; .skip: -> bne X
```

In fact, the `b` itself also has limited range and sometimes needs to be done with the `bl`. See Section 4.3 for evaluation.

3.8 Garbage collector

The STS runtime employs a custom, simple, precise, non-compacting, mark-and-sweep garbage collector. The runtime keeps track of STS parts of stacks of all threads of execution. STS stacks contain only values in the uniform format described above. During collection, pointers in these stacks, as well as in global variables, and in any locations registered dynamically by the C++ runtime are considered live, and recursively scanned for further live pointers. All objects start

with a pointer to v-table, which has a method to determine the size of an object. There is no additional space overhead incurred by the GC: the lowest bit of the v-table pointer is used to mark reachable objects, and the size method[8] is used to walk the heap in the sweep phase.

A per-thread object records the STS stack pointer when an STS function is first called (there can be a C++ stack underneath it), and the stack pointer is stored just before any C++ function is called. If a C++ function in turn calls STS again, which is quite rare, a linked list segment is added to the per-thread object. These per-thread objects are kept by our thread scheduler.

To limit heap fragmentation, we trigger collections more often than strictly necessary. In particular, after every collection we mark the first quarter of free memory as good for allocation. When an object cannot be allocated there, a new collection is triggered, likely moving the location of the "good" quarter. We then allocate the object regardless if it fits in the (new) good part. This risks triggering a bit too many collections, but we have found it to limit heap fragmentation, which was critical when a larger object (eg. a screen buffer) needed to be allocated later in course of the execution. Additionally, given the relatively small size of memory and fast processor speed of the target MCUs, the collections are quite cheap.

The C++ code can either allocate regular GC-able objects to be used on the STS side, or use traditional `malloc()`/`free()` for other purposes. Such malloc-blocks use a special encoding including the size in place of the v-table pointer and are never collected, until they are freed. This heap sharing lets us avoid splitting the memory into C++ and GC heaps upfront. We still keep a small C++ heap to be used inside of interrupt service routines, as the GC cannot be used there.

Arguments to C++ functions called from STS are, in addition to registers, also placed on the STS stack, so that they are not GC-ed, while the function is running. Any allocation can trigger a collection, so intermediate objects allocated in a C++ function, have to be temporarily registered with the GC, before they are returned to STS.

Reference-counting We have previously used reference counting for memory management. We generally incremented reference counts of all arguments to C++ functions for the duration of their execution. This incurred dramatic time overheads compared to the GC described above (see Figure 7).

4 Evaluation

We evaluate STS compiled machine code (as well as the STS virtual machine backend, referenced as VM) on a number of well-known small performance-intensive benchmarks, comparing the performance against:

- a pure C implementation, compiled with gcc, as a baseline for comparison;
- **Duktape 2.3**, an embedded JavaScript interpreter;
- **IoT.js 1.0**, the JerryScript embedded JavaScript interpreter;
- **Python 3.6**, the regular, full-fledged Python interpreter;
- **Node.js 11.0**, which includes V8, a state of the art JIT engine;
- **MicroPython 1.9.4**, an embedded Python interpreter.

We use three different commercially available ARM-based systems for testing:

- **GHI Brainpad**, which uses the STM32F401RE, an ARM Cortex-M4F core with 96kB of RAM and 512kB of flash running at 84MHz;
- **Adafruit Pybadge**, using the ATSAMD51G19, an ARM Cortex-M4F core with 192kB of RAM and 512kB of flash running at 120MHz;
- **Raspberry Pi Zero**, using the BCM2835, an ARM11 core with 512MB of RAM running at constant 700MHz (dynamic CPU frequency scaling was disabled).

The Pi can run Node.js, which places it outside of our target memory range, but we used it for reference comparison of performance against V8. Generally, memory access is much slower on the ARM11 core than on M4F cores (where the entire RAM is similar in performance to L1 cache). The FPU on the M4F cores is only single-precision, so it was not used in benchmarks. The FPU on the ARM11 core was used.

Duktape was compiled with default options on the BCM. On the STM the default profile immediately runs out of memory, so we used the low memory profile (we additionally enabled fast integer option). We used the official ARMv6 Node.js binary. We used Python that comes with PiCore Linux. We used IoT.js compiled by one of its developers (the heap seems hard-set to 512kB, hence the OOM in results). We used the official MicroPython binary for STM32.

4.1 Benchmarks

We used a number of benchmarks, described below. We tried to use equivalent code in TypeScript/JavaScript and Python, as we are comparing run times and not programming languages.[9] The C programs by necessity are not directly feature-comparable, as they lack memory safety and use a static integer representation.

Richards: Martin Richard's benchmark simulates an operating system's scheduler queue. We took JavaScript sources from the JetStream benchmark suite [11] and translated them to TypeScript by replacing prototype initialization with

[8] The indirect call overhead is only 4-8 cycles on a Cortex-M4.

[9] More specifically, Python *fann* was using a different algorithm than the C or JavaScript versions. We changed it to use the same algorithm and direct array accesses. Python *binary* was changed to use classes and not tuples. We made these alterations because we wanted to measure array accesses and class implementation. We did not change *richards* or *nbody*.

	BCM2835 at 700MHz (Pi Zero)							STM32F401RE at 84MHz			
	[ms]	[times slower than GCC]						[ms]	[times slower than GCC]		
Benchmark	GCC	Duktape	iotjs	Python	Node	STS	VM	GCC	Duktape	μPython	STS
binary(7)								25	320	40	2.4
binary(8)	20	40	35	75	8.0	2.5	8	74	OOM	OOM	2.3
binary(9)	40	38	43	78	4.5	2.3	9	149	OOM	OOM	2.1
binary(10)	90	46	67	94	2.8	2.2	11	394	OOM	OOM	OOM
binary(11)	180	50	OOM	91	1.7	2.2	12				
richards(10k)	10	680	490	1210	68	22	113	90	532	287	14
richards(100k)	120	591	400	1048	9	16	94				
fann(8)	20	215	284	415	12	23	136	119	198	166	21
fann(9)	230	190	262	402	3	20	124				
nbody(1k)	3	227	247	170	113	97	213	511	11	6.7	3.7
nbody(10k)	30	230	246	148	14	85	168				
nbody(100k)	310	226	239	143	4	81	167				

Figure 4. Benchmark execution times on BCM and STM32 MCUs. VM is the STS virtual machine. OOM = Out of memory.

equivalent class code, adding type information where necessary. The benchmark uses a number of classes with derived methods for different scheduler tasks, which is meant to test object property access performance. We also used the C and Python versions of this benchmark, which use a single structure for the tasks, together with a switch statement and a function pointer. We developed equivalent TypeScript code (labeled *richards2*), but the class-based version (*richards*) seems to better reflect typical JavaScript coding patterns. The benchmark parameter is the number of scheduling iterations in thousands.

Binary trees: this program comes from The Computer Language Benchmarks Game [10]. Within a loop, it allocates a full binary tree of a given depth, performs a simple computation on it, and then deallocates it. The benchmark is designed to test memory allocation subsystem performance. The C, Python, and TypeScript versions come from the Benchmarks Game. The Python version was modified to use a class, in order to match the TypeScript version closer.

Fannkuch redux: this is another program from the Benchmarks Game that counts the number of specific permutations and tests array access and integer performance. The Python program was modified to use the same algorithm with explicit, single-element array accesses, the way that the C and TypeScript versions do.

N-body: this program is a physics simulation of planets in the solar system, also from the Benchmarks Game, to measure floating point performance.

4.2 STS and VM performance

Figure 4 compares STS performance against various other systems. We present the run time of a C program directly in milliseconds and otherwise as slowdown (number of times) with respect to C.

On the benchmarks heavy on property access (*binary* and *richards*) STS is well over an order of magnitude faster than the interpreters, and less than two times slower than Node.js. Moreover, the interpreters run out of memory much quicker than STS on *binary*. For straight combinatorial computation (*fann*), STS is about an order of magnitude faster than the interpreters but several times slower than Node.js (which employs quite an advanced optimizer in V8). In floating point computation (*nbody*), STS is still significantly faster than interpreters, but 20x slower than Node.js, which utilizes the FPU much better.

Generally, the longer running benchmarks are more indicative of performance. We have also included the results for small iteration counts to be able to compare the BCM and STM directly on the same programs; on such programs the JIT warmup time is significant in Node.js results.

The STS VM is about 5-6x slower than ARM Thumb STS, except for the floating-pointing-heavy *nbody* benchmark, where most of the work is performed in C++ runtime functions. Yet, the VM is still much faster than the interpreters, suggesting that the static memory layout of classes contribute significantly to overall STS performance.

The results on STM32 and SAMD have very small variance (around 0.1%). On BCM we ran benchmarks 10 times and chosen the fastest results.

4.3 Performance of member access

Figure 5 shows measurements of various forms (see Section 3.4) of member access in isolation. The first four rows show the number of cycles needed for a field access on `this` (just an indexed memory lookup), an access with dynamic class subtype check, an access resolved via an interface lookup, and finally an access on a dynamic map object.

For functions, the figure lists a direct call to a procedure, with no type checks, a call into non-virtual class method, a

Access	STM32	SAMD	BCM
`this.field`	6.5	7.5	11.8
`(x as Class).field`	23.2	24.1	35.2
`(x as Iface).field`	52.2	51.6	76.0
`({ ... }).field`	136.9	136.8	156.3
`staticFunction(x)`	14.1	15.1	20.6
`this.nonVirtual()`	14.2	15.1	20.7
`this.method()`	28.3	29.2	41.7
`(x as Class).nonVirt()`	34.4	35.2	45.2
`(x as Class).method()`	34.5	35.2	49.8
`(x as Iface).method()`	57.7	55.4	85.4

Figure 5. Measured cycles taken by various forms of member access on the three MCUs. Each operation was run a million times and average time consumed was converted to cycles.

Operation	STM32	SAMD	BCM
`x = 1`	1.1	1.8	6.3
`x = y`	2.0	1.9	4.6
`x++`	16.2	16.2	20.6
`x += y`	16.2	16.0	22.1
`x *= y`	41.4	41.6	58.7
`x = Math.idiv(x, y)`	56.0	56.1	162.0
`x = {}` (allocation)	212.5	206.0	294.5
`x += y` (double)	414.0	406.5	413.5
`x *= y` (double)	412.0	402.0	442.0
`x /= y` (double)	968.5	963.0	443.5

Figure 6. Measured cycles taken by various language primitives on the three MCUs. Double math operations include allocation and amortized GC (boxing).

call into a virtual class method, and finally an interface call. We also distinguish between calls on `this`, which do not require a dynamic type check, and other non-interface calls that do.

The non-virtual and virtual calls on a variable are dominated by the subtype check, which is performed slightly differently, ending up with almost exactly the same performance. The interface lookup does not need a type check (because the method is looked up in the virtual table of the object on which it is called), which makes it not that much slower than a regular call. The dynamic map lookup depends on the number of fields, but other timings are mostly input-independent.

Figure 7 shows the performance impact of these different ways of accessing members in a full benchmark. The first row (Baseline) lists run times in the default mode, where classes have Java-like virtual tables, in addition to slower interface tables, and fields of classes are accessed by memory lookup at statically computed offset.

The second and third row show slowdowns incurred when methods and fields are looked up via the interface tables. The fourth row shows the slowdown when objects are represented as dynamic maps with linear lookup, in benchmarks that allow for enabling this in a non-invasive way. The slowdowns for field accesses are rather dramatic, on the order of 2x.

The next row show effects of dropping dynamic subtype checks when accessing member of `this` (as its type was already checked when entering the method). Disabling this optimization would allow us to fall back to dynamic lookup also in methods.

The next line compares performance of the simple mark-and-sweep garbage collector to reference counting (see Section 3.8).

Next, we show impact of folding argument conversions into helper functions. It is very small, but it saves 4% of code size of Arcade. Then, we show the effects of the peephole optimizer, which saves 14% of code size on Arcade, and improves performance by a few percent.

The final line measures the impact of all dynamic subtype checks (enabled in Baseline) that we employ to protect the class abstraction. This is the price we pay for following TypeScript semantics, where casts always succeed, but member access may fail.

4.4 Performance of language primitives

Figure 6 shows cycle measurements of various language primitives. As with Figure 5, we timed one million operations of each type and subtracted the time consumed by an empty loop. The time was then converted to main clock cycles for comparison.

The counts are quite similar, but not the same, for the two M4 cores—in both cases programs run from flash, which is much slower than RAM, and the MCUs cache it differently. The BCM numbers are worse, likely because main memory access takes more cycles there, and caches do not always fix it. This is particularly visible for allocation (which includes amortized cost of GC), which then also dominate the floating point operations. Also, there is no hardware integer division on ARM11.

The integer division and multiplication do not currently have a fast path implementation in assembly (due to differences between ARM MCUs), so they are slower than addition, subtraction and bit operations.

4.5 MakeCode Arcade Performance

The MakeCode Arcade handhelds, as shown in Figure 2, are about 100 times faster than the original arcade machines of the 1980s. However, the games in 1980s were programmed in assembly, with heavy use of accelerated graphics (sprites etc.). Arcade, on the other hand, offers very high-level and beginner-friendly APIs in a fully managed, garbage-collected language. As a result, the more complex games (eg., the one

Compiler modification	richards	richards2	bintree	nbody
Baseline	1265ms	1172ms	324ms	1869ms
Methods via interface	13%	0%	4%	0%
Methods+fields via interface	102%	34%	39%	10%
Objects as dynamic maps		143%		
Subtype checks on `this`	19%	0%	4%	1%
Reference counting	165%	128%	100%	64%
Inline conversions	0.1%	-0.7%	0.0%	-0.1%
No peep hole	6%	6%	5%	0.3%
Disable all subtype checks	-12%	-17%	-7%	-3%

Figure 7. Increases in run time when disabling certain optimizations on STM32.

from Fig. 3) are quite playable with STS at 30fps; extrapolating from our benchmarks, an embedded interpreter would run at 1-5fps.

We have also evaluated code size on Arcade games. The STS compiler generates about 37 bytes of ARM Thumb machine code per input (JavaScript) statement, which works out to about 20 bytes per input line of code. Note that unused parts of code are removed, so the game engine is never fully present in the generated binaries. As a comparison, the STS VM, which uses a specialized 16/32 bit encoding achieves about twice the code density.

The C++ runtime occupies about 125kB of flash, while the bootloader and space for internal filesystem take another 64kB, so there is about 320k left for the program and game assets. In practice, this limits the user application (excluding game engine) to 5-10k lines of code, which we have not yet found to be a significant limitation.

5 Related Work

Safe TypeScript [13] and StrongScript [14] both shore up TypeScript's type system with soundness guarantees, backed by runtime checking. Our work is closest to StrongScript, as STS uses a nominal interpretation of classes for code generation and the STS runtime distinguishes between dynamic objects, created with `{ x = ... }` syntax in JavaScript, and class objects, created with `new C(...)` syntax.

STS differs from StrongScript in a variety of ways. First, StrongScript's system guarantees that in the absence of downcasts, a variable of concrete class type `C` is guaranteed to refer to an object of a nominal subtype of `C` or to `null`. STS uses TypeScript's type inference and checking as is, with no modification. In STS, a variable of class type `C` is checked dynamically (upon lookup of a field/method) to determine if its value is a nominal subtype of class type `C`. The check may fail. There is no type checking performed dynamically on arguments. Second, STS erases casts, whereas StrongScript checks them at runtime; instead, STS performs checks at a dereference (member/field lookup), as noted above. StrongScript allows an object of class type to be extended with extra properties. STS currently does not allow such an extension;

Hop.js [17] is a static compiler, with advanced type inference, that translates JavaScript programs to Scheme and then compiles them using Bigloo. Reported performance numbers indicate much better than STS performance on the combinatorial benchmarks (like *fann*) and somewhat worse on *richards* (however, these numbers are from x86 and we estimate from numbers relative to V8; we were unable to run tests on RPi ourselves).

SJS [5, 6] is a similar system, except that it is implemented in Java, and generates C instead of Scheme. Similarly, it seems to exhibit better performance than STS on combinatorial benchmarks, and slightly worse on *richards*. Neither of these is suitable for running in a web browser. In general, it seems these compilers are able to produce very efficient code, when they can infer static types, falling back to much slower runtime path when they cannot. STS, on the other hand, exhibits rather flat and predictable performance. This should allow for combining the approaches in future.

There are other embedded interpreters in addition to the ones we have compared against such as XS7 [21] or Espruino [23]. These seem to have similar performance performance as the ones we have used.

6 Conclusion

Static TypeScript (STS) fills an interesting niche in compilers for embedded systems. Implemented fully in TypeScript itself, the STS toolchain can run in a web browser and produce ARM (Thumb) machine code for a large subset of TypeScript. For efficiency of compiled code, STS relies on a nominal interpretation of classes. Via MakeCode, STS has been widely deployed across a range of devices with small amounts of RAM. Evaluation of STS on a set of small benchmarks shows that STS's generated code is much faster than various embedded intepreters for scripting languages. The largest STS application to date is MakeCode Arcade, whose game engine comprises over 10,000 lines of STS.

Acknowledgments We would like to thank current and former members of the MakeCode team: Abhijith Chatra, Sam El-Husseini, Caitlin Hennessy, Steve Hodges, Guillaume Jenkins, Shannon Kao, Richard Knoll, Jacqueline Russell, and Daryl Zuniga. We also express our gratitude to James Devine and Joe Finney at Lancaster University, the authors of CODAL used as a layer of our C++ runtime. Finally, we would like to thank the anonymous reviewers for their helpful comments, and Edd Barrett for his help in getting the final version of this paper ready.

References

[1] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli de Halleux, Steve Hodges, Michal Moskal, and Gareth Stockdale. 2019. The BBC micro:bit – from the UK to the World. *Commun. ACM (to appear)* (2019).

[2] BBC. 2017. BBC micro:bit celebrates huge impact in first year, with 90% of students saying it helped show that anyone can code. https://www.bbc.co.uk/mediacentre/latestnews/2017/microbit-first-year.

[3] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[4] Hans-J Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: an alternative to strings. *Software: Practice and Experience* 25, 12 (1995), 1315–1330.

[5] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il 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, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 410–429. https://doi.org/10.1145/2983990.2984017

[6] Wontae Choi, Satish Chandra, George Necula, and Koushik Sen. 2015. SJS: A type system for JavaScript with fixed object layout. In *International Static Analysis Symposium*. Springer, 181–198.

[7] James Devine, Joe Finney, Peli de Halleux, Michal Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2018, Philadelphia, PA, USA, June 19-20, 2018*. 19–30.

[8] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. 2015. Ultra Lightweight JavaScript Engine for Internet of Things. In *SPLASH Companion 2015*. 19–20.

[9] Damien George. 2018. MicroPython. http://www.micropython.org.

[10] Isaac Gouy. 2018. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[11] Apple Inc. 2018. JetStream Benchmarks 1.1. https://www.browserbench.org/JetStream/in-depth.html.

[12] Adafruit Industries. 2018. CircuitPython. https://github.com/adafruit/circuitpython.

[13] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 167–180. http://doi.acm.org/10.1145/2676726.2676971

[14] G. Richards, F. Z. Nardelli, and J. Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*. 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

[15] Samsung. 2018. JerryScript. http://jerryscript.org.

[16] Sue Sentance, Jane Waite, Steve Hodges, Emily MacLeod, and Lucy Yeomans. 2017. "Creating Cool Stuff": Pupils' Experience of the BBC Micro:Bit. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, 531–536. https://doi.org/10.1145/3017680.3017749

[17] Manuel Serrano. 2018. JavaScript AOT compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*. ACM, 50–63.

[18] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[19] Artifex Software. 2018. MuJS. https://mujs.com/.

[20] Cesanta Software. 2018. mJS. https://github.com/cesanta/mjs.

[21] Patrick Soquet. 2017. XS7. https://www.moddable.com/XS7-TC-39.

[22] Sami Vaarala. 2018. DukTape. https://duktape.org/.

[23] Gordon Williams. 2017. *Making Things Smart: Easy Embedded JavaScript Programming for Making Everyday Objects into Intelligent Machines*. Maker Media.