MakeCode: Types, Games, and Machine Code

Michał Moskal
Microsoft Research Redmond

Joint work with: Thomas Ball, Peli de Halleux, Abhijith Chatra, James Devine, Sam El-Husseini, Joe Finney, Caitlin Hennessy, Steve Hodges, Guillaume Jenkins, Shannon Kao, Richard Knoll, Chase Mortensen, Galen Nickel, Jacqueline Russell, Joey Wunderlich, and Daryl Zuniga.

QCon, San Francisco, November 2019
Microsoft MakeCode

• Open-source [0] platform for computer science education
• Works everywhere as web app [1]
• Low floor, high ceiling
• First deployed for BBC micro:bit ($15 device; 4m+ units deployed)

[0] https://github.com/microsoft/pxt
[1] https://makecode.com
Demo: MakeCode for micro:bit
Demo – basic micro:bit

- Makecode.com – we have many editors, let’s take micro:bit
- On Microsoft.com but open source, similar model to VSCode
- Flashing heart, event handlers for buttons, shake; deploy
- Radio – transmit acceleration – easy, high-level APIs; deploy
- Add servo instead of plot – use map block
- Switch to TS, see how changes are reflected back
- Find map in core/math.ts – explain about block generation
  - Show class in core/music.ts
  - Show showString in core/basic.cpp
  - Show built-binary.js built-binary.asm
Demo – makecode as a platform

• Go to extensions – these are popular, search for robot, car
• Add gigglebot extension
  • Block category is added
  • Localized in French
  • [skip] Go to readme – show blocks rendering – same as in docs and tutorials
    • Screenshots are evil!
  • In pxt.json it specifies a dependency
    • In dependency look at i2cio – buffer operations, array of [0,0,0,...]
• [skip] Try “dice” tutorial
  • search projects/dice.md in microbit repo – show how it’s written
• [skip] Show translation – community-driven, 1k+ translators
Why should you care?

• IoT is coming. For real this time
  • Arm expects 1e12 devices by 2035; now 5e9+
  • Someone will have to program them
  • There’s 10x+ as many web programmers as embedded programmers
  • 32-bit M0+ start at 32 cents. 8-bit is dying.

• High-level languages are the future of embedded programming
  • Python (CircuitPython, MicroPython)
  • JavaScript (Espruino, iot.js)
  • TypeScript (MakeCode)
Not convinced?

• Rise your hand if you think hiring developers is easy. Anyone?
  • Many more new programming jobs than graduates (universities+bootcamps)
  • Many non-programming jobs will involve programming

• MakeCode is (an example of) the future of *all* of programming!
  • High-level, layered abstractions
  • Just extrapolate: 1101100 -> ASM -> C -> C++ -> Java -> JS -> npm -> ???
  • Very low floor, and reasonably high ceiling

• We pay for abstractions with performance. But how much?
Richard’s benchmark

Back to embedded systems.

List-walking, modification, virtual calls.
Fannkuch redux

Tight inner loop with array operations.

No real optimizer in STS compiler.
Static TypeScript [0]

• Subset of TypeScript (itself a superset of JavaScript)
  • Removes ‘eval’ and prototype inheritance
  • Keeps closures, ES6 classes, GC, exceptions, ‘any’ type
  • Control flow is ‘static’, data handling is often dynamic (i.e., numbers are all conceptually doubles)

• Main target: 32-bit ARM Cortex-M microcontrollers; 16-256kB of RAM

• Decent performance with native compile and custom runtime
  • sometimes in ballpark of V8 and 10x+ faster than interpreters

• Open-source compiler and assembler implemented in TypeScript
  • TypeScript really needed!

[0] https://makecode.com/language
Custom runtime

• Classical Java-like v-tables and field layout in classes
  • With efficient name-based property lookup
• Numbers are tagged 31-bit integers or boxed doubles
  • Math operation hand-coded in assembly
• All strings are length-prefixed UTF8 (and ‘\0’ terminated)
  • Longer non-ASCII strings have additional jump list for faster indexing
  • Cons-strings (ropes) for constant-time concatenation
• Closures capture by value only
  • Mutable locals from outer scopes are allocated as cells on heap
• Simple mark-and-sweep GC (2x faster than previous ref-counting)
Does performance matter?

• Yes. Eventually.
  • Given enough users someone will bump against the limitations
  • They are your power user and they will be vocal about it
  • CPU performance = energy usage

• For micro:bit memory consumption matters.

• In some places, the faster the better.
  • Enter Arcade!
Demo: MakeCode Arcade
MakeCode Arcade demo

• Start at [https://makecode.com](https://makecode.com)
• There are tutorials, videos, community games, etc.
• New project, create duck, makeup, move duck, add gravity, add vx on btn
• Go to flappy duck: only ~60 lines of code – rest is pictures; simplest version around 20
• Go to 3d map
• Download – talk about hardware
  • While downloading talk about the USB drive; deploy
• Show binary.asm, binary.js
• Show game library – hitbox.ts, also sprites.ts
Takeaways

• IoT is coming, it will be programmed in today’s high-level languages

• Programming environments of the future will be even more high-level

• Performance matters. Eventually

• Static types are good for you
The end

• Try it out live at https://makecode.com
• Check out sources at https://github.com/Microsoft/pxt
• Learn more about STS at https://makecode.com/language

• Questions?
Backup
Technical challenges

• Programming environment needs to work offline, in the browser
  • IT admins blocking software installation
  • Spotty internet, scalability, responsiveness, cost, all prevent cloud compilation

• Driver-less deployment from student’s computer to device
  • We developed UF2 file format to simplify that

• Very little RAM (sometimes as little as 2kB)
  • Hard for interpreters, e.g. MicroPython on micro:bit cannot do Bluetooth
  • Can’t run a full-fledged JIT

• It’s difficult to make things simple!
  • Programming language, APIs (also blocks), UI, ...
## Performance on small benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>[ms] GCC</th>
<th>[times slower than GCC]</th>
<th></th>
<th></th>
<th></th>
<th>[ms] GCC</th>
<th>[times slower than GCC]</th>
<th>Duktape</th>
<th>µPython</th>
<th>STS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Duktape</td>
<td>iotjs</td>
<td>Python</td>
<td>Node</td>
<td>STS</td>
<td>VM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary(7)</td>
<td>20</td>
<td>40</td>
<td>35</td>
<td>75</td>
<td>8.0</td>
<td>2.5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary(8)</td>
<td>40</td>
<td>38</td>
<td>43</td>
<td>78</td>
<td>4.5</td>
<td>2.3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary(9)</td>
<td>90</td>
<td>46</td>
<td>67</td>
<td>94</td>
<td>2.8</td>
<td>2.2</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary(10)</td>
<td>180</td>
<td>50</td>
<td>OOM</td>
<td>91</td>
<td>1.7</td>
<td>2.2</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>richards(10k)</td>
<td>10</td>
<td>680</td>
<td>490</td>
<td>1210</td>
<td>68</td>
<td>22</td>
<td>113</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>richards(100k)</td>
<td>120</td>
<td>591</td>
<td>400</td>
<td>1048</td>
<td>9</td>
<td>16</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fann(8)</td>
<td>20</td>
<td>215</td>
<td>284</td>
<td>415</td>
<td>12</td>
<td>23</td>
<td>136</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fann(9)</td>
<td>230</td>
<td>190</td>
<td>262</td>
<td>402</td>
<td>3</td>
<td>20</td>
<td>124</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nbbody(1k)</td>
<td>3</td>
<td>227</td>
<td>247</td>
<td>170</td>
<td>113</td>
<td>97</td>
<td>213</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nbbody(10k)</td>
<td>30</td>
<td>230</td>
<td>246</td>
<td>148</td>
<td>14</td>
<td>85</td>
<td>168</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nbbody(100k)</td>
<td>310</td>
<td>226</td>
<td>239</td>
<td>143</td>
<td>4</td>
<td>81</td>
<td>167</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**STM32F401RE at 84MHz**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>[ms] GCC</th>
<th>[times slower than GCC]</th>
<th>Duktape</th>
<th>µPython</th>
<th>STS</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary(7)</td>
<td>25</td>
<td>320</td>
<td>40</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>binary(8)</td>
<td>74</td>
<td>OOM</td>
<td>OOM</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>binary(9)</td>
<td>149</td>
<td>OOM</td>
<td>OOM</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>binary(10)</td>
<td>394</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td></td>
</tr>
<tr>
<td>richards(10k)</td>
<td>90</td>
<td>532</td>
<td>287</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>richards(100k)</td>
<td>119</td>
<td>198</td>
<td>166</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>fann(8)</td>
<td>119</td>
<td>198</td>
<td>166</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>fann(9)</td>
<td>511</td>
<td>11</td>
<td>6.7</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>nbbody(1k)</td>
<td>30</td>
<td>230</td>
<td>148</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>nbbody(10k)</td>
<td>310</td>
<td>226</td>
<td>143</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>nbbody(100k)</td>
<td>310</td>
<td>226</td>
<td>143</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
## Effects of optimizations

<table>
<thead>
<tr>
<th>Compiler modification</th>
<th>richards</th>
<th>richards2</th>
<th>bintree</th>
<th>nbody</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1265ms</td>
<td>1172ms</td>
<td>324ms</td>
<td>1869ms</td>
</tr>
<tr>
<td>Methods via interface</td>
<td>13%</td>
<td>0%</td>
<td>4%</td>
<td>0%</td>
</tr>
<tr>
<td>Methods+fields via interface</td>
<td>102%</td>
<td>34%</td>
<td>39%</td>
<td>10%</td>
</tr>
<tr>
<td>Objects as dynamic maps</td>
<td>143%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtype checks on <code>this</code></td>
<td>19%</td>
<td>0%</td>
<td>4%</td>
<td>1%</td>
</tr>
<tr>
<td>Reference counting</td>
<td>165%</td>
<td>128%</td>
<td>100%</td>
<td>64%</td>
</tr>
<tr>
<td>Inline conversions</td>
<td>0.1%</td>
<td>-0.7%</td>
<td>0.0%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>No peep hole</td>
<td>6%</td>
<td>6%</td>
<td>5%</td>
<td>0.3%</td>
</tr>
<tr>
<td>Disable all subtype checks</td>
<td>-12%</td>
<td>-17%</td>
<td>-7%</td>
<td>-3%</td>
</tr>
</tbody>
</table>
## Performance of basic operations (cycles)

<table>
<thead>
<tr>
<th>Access</th>
<th>STM32</th>
<th>SAMD</th>
<th>BCM</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>this.field</code></td>
<td>6.5</td>
<td>7.5</td>
<td>11.8</td>
</tr>
<tr>
<td><code>(x as Class).field</code></td>
<td>23.2</td>
<td>24.1</td>
<td>35.2</td>
</tr>
<tr>
<td><code>(x as Iface).field</code></td>
<td>52.2</td>
<td>51.6</td>
<td>76.0</td>
</tr>
<tr>
<td><code>{ ... }).field</code></td>
<td>136.9</td>
<td>136.8</td>
<td>156.3</td>
</tr>
<tr>
<td><code>staticFunction(x)</code></td>
<td>14.1</td>
<td>15.1</td>
<td>20.6</td>
</tr>
<tr>
<td><code>this.nonVirtual()</code></td>
<td>14.2</td>
<td>15.1</td>
<td>20.7</td>
</tr>
<tr>
<td><code>this.method()</code></td>
<td>28.3</td>
<td>29.2</td>
<td>41.7</td>
</tr>
<tr>
<td><code>(x as Class).nonVirt()</code></td>
<td>34.4</td>
<td>35.2</td>
<td>45.2</td>
</tr>
<tr>
<td><code>(x as Class).method()</code></td>
<td>34.5</td>
<td>35.2</td>
<td>49.8</td>
</tr>
<tr>
<td><code>(x as Iface).method()</code></td>
<td>57.7</td>
<td>55.4</td>
<td>85.4</td>
</tr>
</tbody>
</table>
# Performance of math operations (cycles)

<table>
<thead>
<tr>
<th>Operation</th>
<th>STM32</th>
<th>SAMD</th>
<th>BCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>1.1</td>
<td>1.8</td>
<td>6.3</td>
</tr>
<tr>
<td>x = y</td>
<td>2.0</td>
<td>1.9</td>
<td>4.6</td>
</tr>
<tr>
<td>x++</td>
<td>16.2</td>
<td>16.2</td>
<td>20.6</td>
</tr>
<tr>
<td>x += y</td>
<td>16.2</td>
<td>16.0</td>
<td>22.1</td>
</tr>
<tr>
<td>x *= y</td>
<td>41.4</td>
<td>41.6</td>
<td>58.7</td>
</tr>
<tr>
<td>x = Math.idiv(x, y)</td>
<td>56.0</td>
<td>56.1</td>
<td>162.0</td>
</tr>
<tr>
<td>x = {} (allocation)</td>
<td>212.5</td>
<td>206.0</td>
<td>294.5</td>
</tr>
<tr>
<td>x += y (double)</td>
<td>414.0</td>
<td>406.5</td>
<td>413.5</td>
</tr>
<tr>
<td>x *= y (double)</td>
<td>412.0</td>
<td>402.0</td>
<td>442.0</td>
</tr>
<tr>
<td>x /= y (double)</td>
<td>968.5</td>
<td>963.0</td>
<td>443.5</td>
</tr>
</tbody>
</table>
What’s missing?

• prototype inheritance (including monkey patching)
  • Only regular classes are supported
  • No ‘this’ outside of class
  • No ‘.apply’; also no ‘arguments’

• classes are classes
  • can’t dynamically add fields
  • field accesses only work on that class

• modules
  • Namespaces are supported

• ‘eval’

• ‘yield’, ‘await’
  • we have implicit threads/fibers though

https://makecode.com/language
What’s there?

- All basic JavaScript control flow
- Mark-and-sweep garbage collector
- Functions with lexical scoping, also passed as values
- Namespaces
- String templates
- Enums
- Classes with single inheritance, interfaces, object literals
  - Get/set accessors

https://makecode.com/language
What’s there?

• exceptions (throw, try ... catch, try ... finally)
• explicit or implicit use of the any type, union or intersection types
• typeof expression
• delete statement (on object created with {...})
• binding with arrays or objects: let [a, b] = ...; let { x, y } = ...
  • also object destructuring with initializers
• shorthand properties ({a, b: 1} parsed as {a: a, b: 1})
• computed property names ({{[foo()]: 1, bar: 2}}

https://makecode.com/language
Compiler architecture

• Regular TypeScript compiler generates ASTs
• STS compiler does two passes enforcing restrictions and emitting IR
• Custom IR is transformed to:
  • Continuation-passing style JS for the simulator
  • ARM Thumb machine code
  • Custom VM code
  • RISC-V is in the works
• Non-JS emitters’ output is passed through assembler
• Resulting machine code is appended to pre-compiled C++ runtime
  • With some small patching
  • Runtime is pre-compiled in the cloud for a given set of C++ sources
  • C++ functions take usual C types, and compiler inserts conversion to say uint16_t
V-tables

class A {
    x: number
    foo() {}
    bar() {}
}

class B extends A {
    y: number
    foo() {}
    baz() {}
}

this.x -> this[4]
(p as B).x -> p is B; p[4]
(p as any).x ->
    p[0]->iface +
    p[0]->iface[
        3 * p[0]->hashmult]...

12 (size in bytes)
Magic numbers
17 (class ID)
1101023518 (hash multiplier)
GC method pointers...
A.bar (code pointer)
B.foo (code pointer)

16, 30, 12, 18, ... - hash table
3 (index of “x”)  
4 (offset of .x)  
4 (index of “y”)  
8 (offset of .y)  
2 (index of “foo”)  
1 (index of “bar”)  
A.Bar
0 (end marker)
31-bit numbers

- Pointers (memory locations) are integers
  - They are always divisible by 4 => two lowest bits of pointers are always 0
- STS (like many others) uses odd numbers to represent integers
  - integer N is represented by number 2N+1
  - other numbers are allocated on the heap, boxed
- For example, adding two numbers A, B:
  - Check if both A and B are odd
    - If (A-1)+B doesn’t overflow, it’s the result (16 cycles; important to optimize!)
  - Otherwise, convert to floats, do addition and if needed allocate memory for result (~400 cycles)
Strings

- All string data ‘\0’-terminated (for C++ interop)
  - ‘\0’ inside of string still supported
  - UTF8 data always available directly in memory
- ASCII strings (all characters 0-127): length + data
- Short (20 bytes or so) Unicode strings – like above but data in UTF8
  - Length and indexing computed linearly
- Long Unicode strings: length, size, pointer to data + skip list
  - skip list shows byte offset for character offset divisible by 16
- Cons-strings: pointers to two other strings
  - Transformed in-place into skip-list-string on indexing
  - Makes string concatenation constant-time
Closures

• Pointer to code + read only locals from outer scopes
• Variables that are written to after initialization and captured are transformed everywhere into a pointer to a heap location that holds the actual value
• Functions not capturing anything are allocated statically in the flash
C++ interop

• C++ functions can be exported to TypeScript (and blocks)

• Compiler generates conversions from internal representations to usual C++ types

// source C++ code:

```cpp
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 micro:bit Orchestra

https://www.youtube.com/watch?v=5tX93t7jCbQ
**UF2 – USB Flashing Format**

- Micro:bit has a special chip to talk USB, understand FAT and .HEX format, and flash the target chip
- Most dev boards have a single chip that talks USB (cheaper!)
  - Part of flash reserved for bootloader that can update the main app
  - Bootloaders typically talk custom protocols
- UF2 makes it easy to write a bootloader that talks USB mass storage
  - UF2 files consist of 512-byte self-contained blocks, independent of filesystem

```c
void handle_write(UF2 *b) {
    if (b->magic0 == UF2_MAGIC0 && b->magic1 == UF2_MAGIC1)
        write_flash(b->target_addr, b->data, b->size);
}
```

- Open source at [https://github.com/microsoft/uf2](https://github.com/microsoft/uf2) and widely adopted
Performance examples

• Classes vs dynamic objects – see when it runs out of memory
  • https://makecode.microbit.org/_4y6Yeyh1zbU2

• Adding numbers – 31 bit integers vs floats
  • https://makecode.microbit.org/_hKAPqwHcgbee
Recap: What are we doing here?

• Getting kids excited about computer science
  • Tangible devices, games

• Making embedded programming radically simpler
  • Arduino was great in 2005, but things have moved on

• Letting users under the hood with TypeScript
  • Most of runtime implemented in MakeCode itself
  • Still need better Blocks->Text transition