Pushing Blocks all the way to C++

Jonathan Protzenko
Microsoft Research
One Microsoft Way
Redmond, Washington 98052
Email: protz@microsoft.com

***Abstract*—The BBC `micro:bit` project aims to teach programming to every 11 to 12-year-old in the UK, through the means of a programmable device half the size of a credit card. The device will be freely handed out to every student.**

Microsoft's TouchDevelop programming environment was picked to provide the programming experience for kids; we retrofitted the website for the `micro:bit`. TouchDevelop remains a complex beast: in order to make it easier for 7th graders to program, we added an alternative, visual code editor based on Google's Blockly [1].

This paper is an experience report about the various challenges we met when trying, at one end, to expose a visual Blocks-based programming model, while at the other end generating C++ for the device.

I. Overview of the project

The microcontroller will be handed out to students in the fall, and is about 4cm by 5cm wide. It is equipped with an ARM Cortex-M0 processor, 16k of RAM, 128k of flash memory, a compass, an accelerometer, a Bluetooth Low Energy (BLE) chip, and a series of pins to enable Arduino-style programming. The system features a 5x5 LED array and two buttons for input-output. The chip is programmable via the ARM mbed technology: upon plugging it into a computer, the chip masquerades as a USB key, meaning that it suffices to drag and drop a hex file onto the drive to perform flashing.

The BBC **`micro:bit`** is programmed using either ARM Cortex-M0 assembly or C++. The blessed toolchain is ARM's mbed platform, which consists of a web-based, C++ online IDE. Programs written in the online IDE are compiled in ARM's cloud; the resulting hex file appears as a browser download. We do not expect students to master the subtleties of C++; therefore, students get to write programs using TouchDevelop. Upon hitting "compile", the student's TouchDevelop program is translated to C++, which is then sent over to ARM's cloud. The resulting hex file similarly appears as a browser download.

The stated goal of the project is to teach the basics of programming to all students; as one BBC executive so accurately put it: "if we only manage to teach them that numbering starts at 0 in computing, we'll save the country a lot of bugs already". Joke aside, we expect advanced students to master basic control structures (conditionals, loops); variables; abstraction (via functions). While the **`micro:bit`** enables sophisticated projects that bundle sensors and displays over I2C, we expect most students to play with the onboard compass and accelerometer only.

TouchDevelop is a JavaScript-inspired, statically-typed, syntax-directed, web-based programming language. While suited to beginners, TouchDevelop still makes it possible to write faulty programs that require user intervention to fix. Therefore, in order to lower the entry barrier for 7th graders, we decided to include, in addition to the classic "TouchDevelop editor", an editor based on Google's Blockly. There are many reasons for this: we expect teachers to be more familiar with Blocks-based programming environment, due to prior experience with Scratch or Hour of Code; we also expect students to "click" better with Blocks programs.

The compilation scheme of "Blocks" programs is as follows: they are first (invisibly) translated to TouchDevelop, then the regular compiler from TouchDevelop to C++ kicks in and performs the rest of the job.

Going through TouchDevelop presents two major advantages. The first one is purely technical: we don't need two separate compilation paths. The second one is pedagogical: the student can choose to perform the conversion manually at any time. This presents a learning opportunity, showing the translation from Blocks to a more traditional programming language; it also allows the student to take their program to the next level and "unlock" more programming possibilities.

II. Programming model

One of the challenges we had when designing our Blocks language was to keep a programming model that we believe is simple enough for students to understand, while still making sure we can map Blocks programs to suitable C++. Here is an overview of the programming model we came up with.

Memory model Memory management is automatic; students do not have to allocate or free memory. Scalar values such as booleans and integers are passed by copy. Heap-allocated values such as 5x5 images and character strings are passed by reference.

Threading model Threading is cooperative, meaning that at certain program points, the current thread *yields* control back to the scheduler, hence giving other threads a chance to execute. Examples of threads include forever blocks (which fork a new computation) and event handlers.

Event model Students can either do active polling ("if button A is pressed, then do…"), or register event handlers ("when button A is pressed, do…"). Again, this is fairly convenient for the student, but makes our compilation

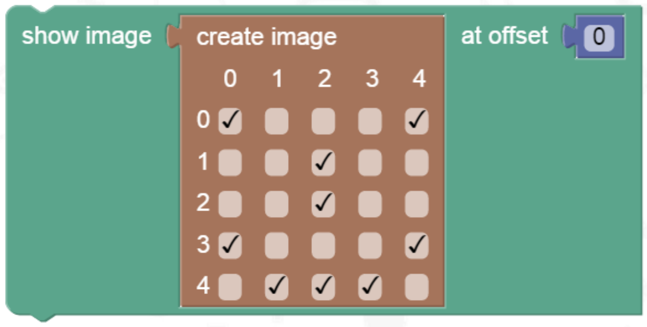

Fig. 1. User-friendly UI for designing new images

scheme more involved, as it involves closures under the hood.

Syntax model Thanks to a careful design of our blocks and their connections, issues such as starving are mitigated. Furthermore, we added some friendly blocks that allow editing 5x5 images and animations easily (Figure 1). It turns out in practice that starving is not an issue.

Type system For performance reasons which we expose below, we wish to statically type Blockly variables in order to generate efficient C++. Therefore, and quite against the Blockly spirit, we require variables to be typed. Types are inferred via a ML-style, Hindley-Milner type inference algorithm [2]. The type inference algorithm has been written outside of the Blockly codebase, meaning that it is not well integrated. Indeed, the user may get an unfriendly error message rather than visual clues. We hope to improve this.

A. Discussion of the programming model

We briefly contemplated skipping type inference, requiring instead that variables be annotated with their types at definition-time. The engineering effort would have consisted of an extra "definition" block, perhaps with some customization of Blockly to ensure the block is always provided[1]. We felt, however, that one of the great strengths of Blockly is its simplicity, and that requiring the user to understand and provide types for variables would have been a deterrent. In the current design, unless the student intentionally mixes, say, numbers and images, the compilation proceeds silently. We do perform a lot of work under the hood, but this remains invisible for the student.

Cooperative threading turned out to be essential: threads are never interrupted, meaning that we completely rule out data races. The main drawback to cooperative scheduling is that threads must yield often enough to avoid starving. In practice, we haven't had starving issues.

An alternative design would have been pre-emptive scheduling, with a hardware interrupt that runs at periodic intervals and schedules another thread for execution. A pre-emptive scheduler (as is found in most modern systems) would expose race-conditions, when two threads try to access the same variable. This would, in turn, take us towards the difficult problem of inserting locks automatically. Locks come with a memory and performance cost. We were happy to let students ignore all these issues.

Reference counting fails in the presence of cycles. It is conceivable that an advanced student indeeds succeeds in creating cycles, using advanced features of TouchDevelop (this is not possible with Blocks). However, ruling this out requires significant more run-time overhead than basic reference counting. We felt that the current tradeoff was satisfactory, and that this was a good learning opportunity for the few highly-advanced students.

[1]BlocklyDuino [3] adopts this approach, but does not seem to check that a variable is always properly defined.

III. The Blocks compiler

The compilation of Blocks, as we mentioned earlier, implies compiling first to the TouchDevelop language. There is nothing profound about choosing TouchDevelop as a target: it just so happens that we already needed a compiler from TouchDevelop to C++; furthermore, we wished to allow the user to perform conversion manually ("graduate"), so a compiler from Blocks to TouchDevelop definitely made sense.

Conjecture 1 (Correctness of TouchDevelop compilation)**.** *If a TouchDevelop program P is well-typed, then the result of compiling P to C++ compiles without errors.*

The semantics of TouchDevelop programs in the context of the **`micro:bit`** are fuzzy. We thus do not state any result about the preservation of semantics.

We of course want to make sure that a Blockly program never generates an (unscrutable) C++ compile error. Therefore, *compiling a Blockly program should generate a well-typed TouchDevelop program*, hence ensuring that compilation succeeds without errors.

Conjecture 2 (Correctness of Blockly compilation)**.** *If a Blockly program P is well-typed, then the result of compiling P is a well-typed TouchDevelop program.*

A. Alternative designs

We chose to generate a statically-typed C++ program that faithfully implements the original Blocks program, with TouchDevelop as an intermediary step. Some other designs are possible.

- We could forgo static typing altogether, and adopt a memory representation where all objects are tagged with their type, and have run-time checks for every operation to ensure that the operands are of the right type. This is dynamic typing. We ruled out this approach for efficiency and memory consumption reasons (the implementation of the BLE stack uses a significant chunk of memory, leaving very little of the original 16k available).
- We could compile Blocks programs to bytecode, and flash the device with a bytecode interpreter along with the bytecode that corresponds to the original program. This

is the approach used by MicroPython [4] and OCaPic [5]. We didn't have the engineering resources to design a bytecode format, and write an interpreter and run-time system for it (including a GC, most likely in Cortex-M0 assembly). Furthermore, our approach minimizes the memory and performance overhead in our constrained context.

B. Type-checking Blockly programs

Programs written in Blockly are dynamically-typed and all variables live in a common, global scope. TouchDevelop is a JavaScript-inspired, statically-typed programming language equipped with lexical scope. In order to successfully convert from the former to the latter, one needs to resolve the types of the program variables. We implemented *type inference for blocks*.

In Blockly, one can, and will want to define custom blocks. For instance, we have a "show image" block that displays an image on the 5x5 LED array. When defining a custom block, one specifies its shape; color; connections; labels, and so on. One can optionally provide types: for instance, the "+ (arithmetic plus)" block takes two numbers and returns a number. Similarly, the "show image" block takes a **micro:bit** image and returns nothing. These type annotations are leveraged by the user interface of Blockly: if the user tries to fill the "show image" block with a number, the smaller block "pops out" of the outer one, and the user cannot, "syntactically", write buggy code (Figure 2).

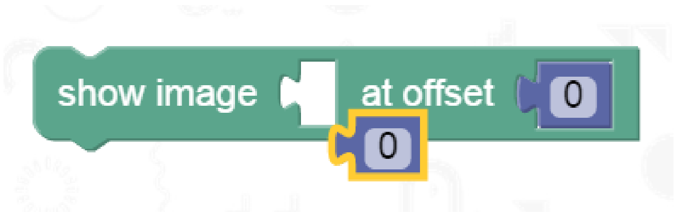

Fig. 2. Blockly's UI prevents connecting these two blocks: the "show image" block is defined as taking an image, not a number.

Not all blocks can be annotated, though. Since the types are static, and provided at *block-definition time*, one cannot provide a proper type annotation for the variable block. Therefore, the Blockly user interface performs no check for this block, meaning that invalid (per the TouchDevelop semantics) programs can be written, such as "show image item", where "item" has been assigned a number earlier (Figure 3).

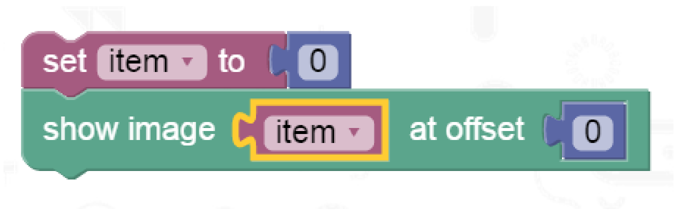

Fig. 3. This program cannot be translated to valid C++, since the `item` variable cannot be both an integer and an image.

In order to target a statically-typed language (TouchDevelop), which will in turn allow us to generate efficient C++ (with no dynamic types), we must infer the type of Blockly variables.

This is the classic type inference problem, wherein each block has an expected type for its arguments, and a return type. For instance, we may understand the "+ (plus)" block to have signature `int plus(int x, int y)`. Some blocks have a polymorphic type: for instance, equality comparison is meant to take two operands of the same type, so one may understand it to have type `bool equals<T>(T x, T y)`. Our type-checking algorithm therefore must perform unification, where upon comparing `x` and `y` for equality, we unify their types.

The algorithm kicks in at compile-time (or graduation-time). Since all variables are global, it suffices to loop over all blocks in no particular order. The type-checking algorithm is easy, since there is no subtyping or structural types such as pair or records (we did not enable lists). For each block of the program, we look up its associated typing rule and determine types accordingly. An English, readable version of the rules may be as follows:

- when assigning to `i`, the type of `i` must be the same as the return type of the right-hand side;
- when comparing two blocks with "=" (equality), the type of the two operands should match;
- when summing two blocks with "+" (plus), both operands should be numbers;
- etc.

It may be the case that the type of a variable still remains undetermined after type inference. This is not an error; this happens, for instance, when a variable has been defined, then never been assigned to. In this situation, we arbitrarily choose "number" for the type.

The algorithm uses textbook, imperative union-find structures to perform unification. Type inference and code generation are implemented as two separate phases. The implementation is made easier by the fact that there is no lexical scope, meaning that we don't need to maintain a lexical environment and deal with shadowing issues.

Because of the relative simplicity, the only possible issue is a type mismatch between the expected type and the actual type. We generate a classic message of the form "this variable is a number, but we wanted an image". We then highlight the faulty blocks in the user interface with extra CSS classes (Figure 4).

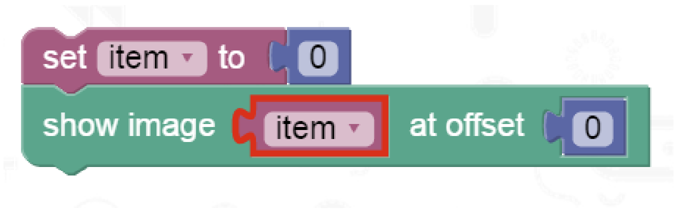

Fig. 4. Type inference flags the faulty program; in this picture, the `item` variable block is highlighted in red.

C. Run-time system

A C++ run-time system written by the University of Lancaster implements lightweight cooperative threading using fibers. In essence, multiple "threads" compete for execution. It is up to the current thread to yield control back to the runtime. This happens when: pausing, doing IO (reading the status of the pins, polling the buttons), scrolling images…

The only possible way to starve other threads is to write a **while** (true) loop and not use the run-time system within the loop body. In our context, this basically amounts to a useless loop; only bogus programs would have such an issue. Therefore, starving has not been an issue for us.

An interesting issue is the "forever" block. The block runs its body in a loop; however, implementing "forever" as **while** (true) would preclude any subsequent instructions from executing. It turns out that the intuitive semantics of the "forever" block is for the body to be executed in a loop, *on a new fiber*. Students naturally write several such "forever" blocks in the workspace and expect them to all run in "parallel". Our implementation makes sure that the fiber yields at regular intervals (Figure 5).

```
function forever (body : Action) do
  /* Repeat the code forever in the
     background. On each iteration, allows
     other fibers to run. */
  control -> in background do
    while true do
      body -> run
      basic -> pause(20)
    end while
  end
end function
```

Fig. 5. The implementation of the forever combinator

This raises the issue of forking: one could write a program that creates an unbounded number of new fibers, hence crashing the run-time system. In order to mitigate the fork-bomb issue, blocks that result in the creation of a new fiber ("forever", "on button pressed") cannot be nested within other blocks (they have no top connection). This means that the fork-bomb "forever (forever …)" cannot be written in Blocks (it can, though, in TouchDevelop).

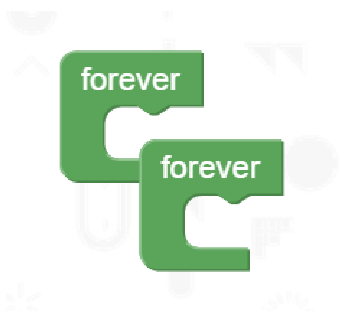

Fig. 6. The forever block is crafted in such a way that it cannot be nested

Having several fibers running at the same time is actually fairly useful: one may want to poll some output pin in the background (Makey makey-style project), while blinking an LED at a regular interval. Such a situation is easily expressed with two cooperative fibers.

D. Semantic matches and mismatches

Writing "when button A is pressed do…" essentially amounts to writing a closure, where the event handler may capture to variables in scope. The good thing is, in Blockly, all variables are global. Translated Blockly programs thus never refer to local variables, which completely eliminates the issue of proper scoping of captured variables, and of their subsequent compilation using a garbage-collected, heap-allocated block. This is an area where Blockly's model of global variables proved beneficial for us.

A difficulty was the compilation of for blocks into TouchDevelop for-loops. TouchDevelop only features a specific form of for-loop, where i is immutable, and one writes `for 0` $\leq$ `i < ...` and only specifies the upper bound. This means that we had to replace the original Blockly for-loop (remove the "step" and "lower bound" parameters), in order to better match the TouchDevelop for-loop. Still, one cannot always translate a Blockly for-block into a TouchDevelop for-loop: Blockly still allows one to assign to the index (anywhere) or read it (outside the loop). We thus had to implement a series of checks (and reconstruct a notion of lexical scope in Blockly) to determine whether a Blockly for-block would result in a TouchDevelop for-loop or while-loop. This is partly due to our own constraints (we insisted on targeting TouchDevelop), party due to the very lenient model of Blockly.

IV. Looking forward

Arduino-like, hardware-based programming projects are gaining momentum. This is, after all, well deserved: unlike deploying a website to the cloud, one only needs to learn one language, the result is immediate, and the student can easily relate their program to the observed output.

One key feature in the **micro:bit** project is that it requires no special software: programming happens in the web browser, compilation happens in the cloud. Therefore, we believe that there is great potential in marrying blocks- and web-based programming environments with hardware targets.

In that context, blocks-based programming environments would benefit from a stricter, opt-in discipline that makes the rest of the compilation easier. One could, for instance, integrate the type-checking discipline within the Blockly codebase, and provide visual feedback based on the type of variables ("pop-out" on type mismatch, use colors…).

From the developer's perspective, facilities such as testing whether a variable is lexically scoped *within* a block would also be beneficial, as we could allocate some variables on the stack in the resulting, translated program.

References

[1] N. Fraser *et al.*, "Blockly: A visual programming editor," 2013. [Online]. Available: https://developers.google.com/blockly/

[2] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the american mathematical society*, pp. 29–60, 1969.

[3] F. Lin, "BlocklyDuino, a web-based editor for Arduino," 2012. [Online]. Available: http://gasolin.github.io/BlocklyDuino/

[4] D. George, "Micro python," 2013. [Online]. Available: https://micropython.org/

[5] B. Vaugon, P. Wang, and E. Chailloux, "Programming microcontrollers in OCaml: The OCaPIC project," in *Practical Aspects of Declarative Languages*. Springer, 2015, pp. 132–148.