Programming microcontrollers through high-level abstractions: The OMicroB project

Steven Varoumas, Basile Pesin, Benoît Vaugon, Emmanuel Chailloux

Highlights

Programming Microcontrollers through High-Level Abstractions: the OMicroB Project

Steven Varoumas, Basile Pesin, Benoît Vaugon, Emmanuel Chailloux

- High-level programming for low resources hardwares

- Virtual machines and runtime environments

- Embedded and cyber-physical systems

- Multiparadigm languages

Programming Microcontrollers through High-Level Abstractions: the OMicroB Project

Steven Varoumas[1]

Huawei Technologies Research & Development UK, Cambridge, United Kingdom.

Basile Pesin

INRIA Paris, Parkas, France.

Benoît Vaugon

Armadillo, 92170 Vanves, France.

Emmanuel Chailloux

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France.

Abstract

In this paper, we present an approach for programming microcontrollers that provides more expressivity and safety than the low-level language approach traditionally used to program such devices. To this end, we provide various abstraction layers (abstraction of the microcontroller, of the electronic components of the circuit, and of concurrency) which, while being adapted to the scarce resources of the hardware, offer high-level programming traits for the development of embedded applications. The various presented abstractions make use of an OCaml virtual machine, called OMicroB, which is able to run on devices with limited resources. These take advantage of the expressivity and safety of the OCaml language (parameterized modules, advanced type system). Its extensibility allows to define a synchronous extension to manage concurrency while keeping a good level of efficiency at execution. We illustrate the value of our work on both entertainment applications and embedded software examples.

Keywords: Microcontrollers, Virtual machine, High-level programming, OCaml, Functors, Synchronous programming

1. Introduction

The diversity of the components of the Internet of Things (IoT) implies several levels of knowledge and technologies on sensors, communications, cloud and user services, which require different

---

Email addresses: steven.varoumas@proton.me (Steven Varoumas), basile.pesin@inria.fr (Basile Pesin), benoit.vaugon@gmail.com (Benoît Vaugon), emmanuel.chailloux@lip6.fr (Emmanuel Chailloux)

[1]This work was carried out when the author was member of LIP6, Sorbonne Université, Paris, France.

programming skills. In this article we focus on embedded systems, in particular on low-resources microcontrollers, their interactions and communications.

Microcontrollers (MCU), and more specifically microcontrollers without operating systems, are devices that can be quite difficult to program. Developers for this kind of hardware traditionally use low-level programming languages (such as C or assembly languages) in order to fine-tune the power and memory consumption of a program, as well as its hardware-specific interactions with its environment. This leaves programmers who are less well-versed in embedded programming with a steep learning curve for developing basic applications, as a lot of hardware documentation needs to be assimilated to design even simple programs. Moreover, debugging such applications is a tedious task, with many programmers (even professionals) relying on executing their application directly on the intended hardware to test its behavior. This can be time-consuming, may not guarantee the correct execution of the application, especially if concurrency is involved, and may cause electrical damage to the hardware.

In order to free developers from needing to program microcontrollers using low-level languages and consequently limited abstractions, multiple projects using a high-level programming language for developing microcontroller applications have been undertaken. These projects provide more expressive programming constructs for the design of embedded software which allow developers from various backgrounds to more easily write applications. Many of these projects involve running a subset of a general purpose programming language on specific microcontrollers, such as Java [1] for object languages or Python [2] and Scheme [3] for dynamic languages. Different criteria are privileged according to the projects and the targeted microcontrollers. It may be the size of code and the runtime memory used, the efficiency on specific architectures, the ease of use of dynamic languages, the expressiveness, the portability to different targets and the safety by using static typing and concurrency models. Of course, these different criteria can be mixed, in particular by following a virtual machine (VM) approach which allows to stay small (as the Scheme VM described in [4]) with a runtime library (for automatic memory management), to accept expressive languages and to be portable while keeping good efficiency.

A key motivation of our work is to provide embedded system developers with a way of leveraging the full set of features of a modern high-level language in order to ease the programming process as well as provide increased guarantees for the safety of programs. To this end, we elected to use the OCaml programming language, a statically typed multi-paradigm language featuring functional, imperative, modular, as well as object-oriented traits. The core of the language is functional, with the combination of ADT (algebraic data type) and pattern matching allowing case-based definition of functions, with static analysis of pattern completeness detection. A short introduction to the language, called OCaml for the masses [5], explained ten years ago "Why the next language you learn should be functional" by following a company's switch to OCaml. The increased expressivity provided by these diverse programming paradigms is strengthened by another important feature of the OCaml language, which is its strong static type system. The OCaml type system helps programmers develop high-level applications by detecting typing errors at compile-time rather than runtime. This detection of errors ahead of the program execution represents a notable benefit when developing embedded programs, where a malfunction can cause harm. Furthermore, the OCaml type system relies on type inference, which gives programmers the option remove type annotations from their programs and focus on its logic and algorithmic aspects, while keeping the same level of type safety. The OCaml language offers two compilation paths: a native version for efficiency targeting the specific architectures of current processors, and a version for portability producing

bytecode for its virtual machine (VM). The quality of its compiler has just been recognized with the prestigious 2023 ACM SIGPLAN programming languages software award.

OCaml developers benefit from a rich ecosystem of tools and libraries, most often distributed by the opam package manager. OCaml's preferred domain is software dependability, with tools for verifying program properties (proof assistant, static analyzer, abstract interpretation, model-checking), certified compilers, or industrial synchronous programming, but other sectors have emerged over time, such as finance, systems and distributed programming (file synchronizer, social networking . . .), as well as entertainment applications.

Although the OCaml language is used for the production or verification of embedded code, OCaml programs themselves are seldom embedded, but we believe that the high-level features of the language are welcome in the context of developing programs for embedded systems. In this context, a first experiment, called OCaPIC [6], was carried out targeting a very low resource architecture for the OCaml language. To achieve this, OCaPIC is based on a 16-bit OCaml VM (sometimes called ZAM [7]) that was implemented in the PIC18 assembly language. It comes with a specific runtime library, also implemented in assembler, and a modified standard library. The design has been experimentally validated by several hand crafted implementations. This work is described in [6] which details the different space saving techniques used. This experiment was successful, but the fact that OCaPIC was built in a fine-tuned PIC18 assembler makes it compatible with just microcontrollers of this family, limiting its portability. We thus endeavoured to keep the ideas of several memory optimizations while offering a solution that can be used for a greater variety of microcontrollers.

In this paper, we present an approach more portable than previous experiments, capable of running programs using any feature of the OCaml language on devices with very little resources. This approach is based on a new implementation of an OCaml virtual machine (VM) that is easily portable across many architectures, and at the same time compatible with microcontrollers with as little as 2.5 kB of RAM. By doing so, we aim to offer to embedded systems developers every benefit of using the OCaml language, including its rich multi-paradigm programming model, and its static type system, to increase the expressiveness and safety or programs, all the while keeping memory footprint low. This portable approach also offers new hardware abstractions that are welcome in a context where hardware architectures are very diverse, making use of the extensibility of the OCaml language allows us to further abstract many aspects of embedded programs. To do this, we take advantage of parameterized modules (functors) and high level typing paradigms like polymorphic variants and Generalized Algebraic Data Types (GADT).

One of the main difficulties comes from the concurrency of programs. Since typical microcontroller programs simultaneously manage multiple hardware entities that share data, they are concurrent by nature. There already exist extensions for reactive programming, in particular in the style of functional reactive programming such as [8] and [9]. These programming features are widely used for critical embedded systems, as shown by the Scade software suite [10]. Many models exist but for embedded applications, synchronous programming [11] brings advantages of simplicity and safety. We thus define a synchronous dataflow extension to OCaml, *à la* Lustre [12], called OCaLustre.

This paper presents a succession of abstraction layers that increasingly provide new paradigms and heightened guarantees to the programming of microcontrollers. Following the plan of [13] to go up in abstraction, it gives a mature point of view on the OMicroB project by detailing precisely

3

its principles and its implementation, and by enriching it with the latest evolutions. In Section 2 we thus describe our OCaml VM called OMicroB, and in particular the different steps taken to convert an OCaml source code into an executable program for a microcontroller. In Section 3 the powerful mechanism of OCaml's functors is used in order to propose a declarative way of describing the electronic circuit of a given application. This solution can be seen as a further hardware abstraction that increases portability and expressivity of our approach. In Section 4, we make use of the extensibility of the OCaml language to implement a synchronous programming extension to OCaml. This synchronous dataflow language offers an implicit model of concurrency that enjoys a new layer of abstraction for the concurrent aspects of a microcontroller's application. Every solution presented in this paper is accompanied by a practical example. Finally, we discuss in Section 5 our positioning relative to other projects dedicated to using high-level languages for programming microcontrollers. We conclude this article with an overview of current and future works dedicated to providing even more layers of abstraction and guarantees for programming microcontrollers.

The main advantages and contributions of our work are:

- **Portability** and **safety** of microcontroller programming thanks to the use of an OCaml virtual machine that can be easily adapted to various architectures.
- **Compositionality** of programs thanks to the abstraction of the electronic components of a circuit interacting with the microcontroller.
- A **lightweight** and **safe model of concurrency** provided by a synchronous extension to OCaml.

Each of these contributions constitute a high-level abstraction layer for programming microcontrollers (see Fig. 1).

2. Abstracting Hardware: The OMicroB Virtual Machine

In order to enhance the portability, as well as the expressivity and the safety of microcontroller programming, we developed OMicroB[2], a specialized implementation of the OCaml virtual machine (the *ZAM - Zinc Abstract Machine* [7]) that is designed to run on microcontrollers with limited resources. This implementation can indeed run non-trivial OCaml programs on small microcontrollers, and provides the embedded software developer with all the high-level programming paradigms of the OCaml language (functional, imperative, modular, object-oriented) and an increased safety through static typing and automatic memory management. Being written in standard C, this virtual machine (and its standard library) is highly portable, as we consider that the C language can be seen as a *portable assembly language* since most microcontroller architectures come with powerful enough C compilers. While our virtual machine approach has previously been described for AVR microcontrollers [14], this work consists in a maturation of the OMicroB virtual machine with an extension of its runtime with more powerful and fine-tuned features (such as a new garbage collection algorithm and the handling of interrupts and callbacks). This has allowed porting OMicroB to new families of devices (ARM, PIC32) and microcontrollers (Micro:bit 2), and supporting evolutions of the OCaml virtual machine itself.

---

[2]Open source repository of the project: `https://github.com/stevenvar/OMicroB`
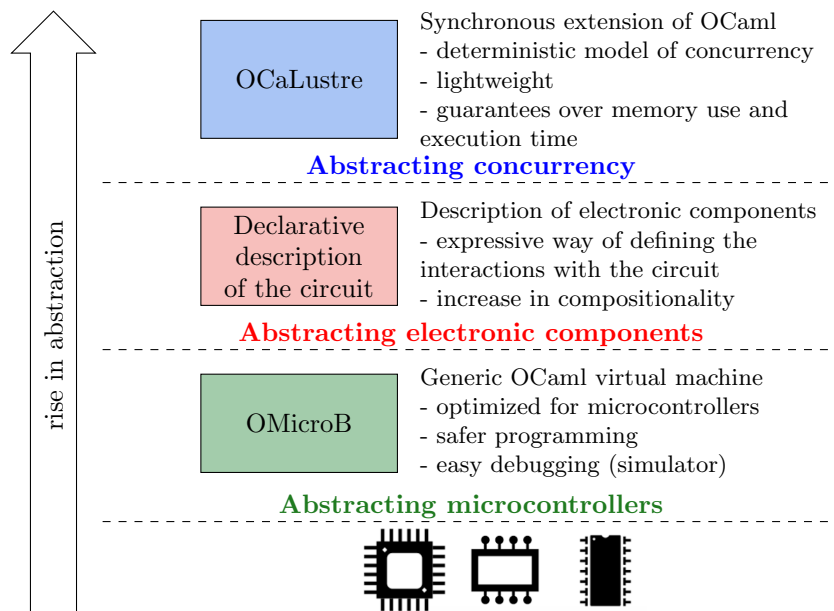
Figure 1: High-level abstractions provided by this work

2.1. Bytecode Interpreter and General Application Mechanism

One central component of OMicroB is a bytecode interpreter that is able to handle the entire bytecode instruction set produced by the standard OCaml bytecode compiler (`ocamlc`). ZAM is a functional virtual machine with eager evaluation and has a relatively small footprint with its 149 instructions, including imperative features, exceptions and method call. Its main memory areas (code, heap, stack, global values) are described in Figure 2. This stack-based interpreter, which uses multiple registers (an accumulator `acc`, a stack pointer `sp`, a program counter `pc`, a pointer to the highest exception handler in the stack (`trap_sp`), etc). As described in Fig. 2, it notably performs the same optimized general application mechanism as the ZAM for n-ary function application, by using a specified `extra_args` register that counts the number of additional arguments given to a function.

This mechanism prevents the VM from currying every function and systematically creating a new closure (represented by a classical environment-code pair) everytime a function with an arity of more than 1 is applied, which is costly. For this, a `GRAB` bytecode instruction is positioned before the instructions corresponding to the body of a function. This instruction deals with the three cases of application by checking `extra_args`: total application, where the function is given as many arguments as it has parameters; partial application, where the function is given less arguments than its number of parameters (in that case, a new closure is created); and the case where there are more arguments than parameters, when the body of the function returns a closure that is immediately re-applied to the remaining arguments.

We illustrate this application mechanism using the code shown in Table 1. In this figure, the top-left code snippet defines an OCaml function called `f` that adds together its three parameters `x`, `y`, and `z`. The bottom left part of Table 1 shows the bytecode generated for this function. When `f` is applied, the `GRAB 2` instruction checks if `2` (two) parameters are given in addition to the
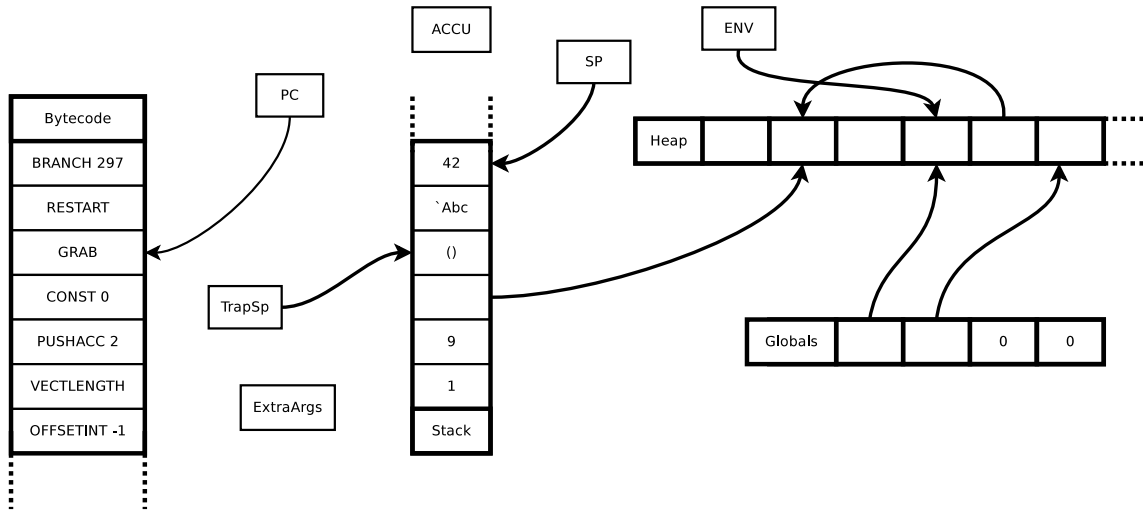
5

Figure 2: Representation of ZAM memories

let f x y z = x + y + z	**let** res = f 4 5 6

```   RESTART```
 ```L1:GRAB 2```
 ```   ACC 2```
 ```   PUSH```
 ```   ACC 2```
 ```   PUSH```
 ```   ACC 2```
 ```   ADDINT```
 ```   ADDINT```
 ```   RETURN 3```
	``` CLOSURE L1, 0```
	 ``` PUSH```
	 ``` CONST 6```
	 ``` PUSH```
	 ``` CONST 5```
	 ``` PUSH```
	 ``` CONST 4```
	 ``` PUSH```
	 ``` ACC 3```
	 ``` APPLY 3```

Table 1: Generated bytecode for `f` and its application

first parameter, and if so directly evaluates the function without creating intermediate closures. Otherwise, a closure is created with an environment containing the arguments in the stack and a code address pointing to the previous RESTART instruction (which copies the closure environment to the stack and sets extra_args to the number of applied arguments when the function is called). The body of the function is constituted of multiple ACC 2 and PUSH instructions that each time copy the third element (indexed by the number 2) of the stack (i.e. a parameter of the application) into the accumulator register and push it to the top of the stack; and two ADDINT instructions that add one value popped from the stack with the value inside the accumulator. The RETURN 3 instruction pops three elements from the stack. Then, if the function returns a closure and there are still arguments to be applied, it goes on with this application; otherwise execution jumps back to the caller of `f` and the saved state of the caller (code pointer, environment, extra_args) is popped from the stack.

The code on the right corresponds to an application of the `f` function to three arguments. It first creates a closure corresponding to the function with CLOSURE, provided with a pointer to the code of the closure (L1) and a 0 to represent the absence of captured variable in the closure, and

then pushes the arguments on the stack. Finally, it calls the function with the three arguments present in the stack via the APPLY 3 instruction, saving its current state in the stack and jumping to the bytecode located at label L1.

Table 2 shows the evolution of the content of the registers of the virtual machine during the execution of this example program.

instruction	accumulator	stack	environment	extra_args
CLOSURE L1, 0	L1	[]	[]	0
PUSH	L1	[L1]	[]	0
CONST 6	6	[L1]	[]	0
PUSH	6	[6, L1]	[]	0
CONST 5	5	[6, L1]	[]	0
PUSH	5	[5, 6, L1]	[]	0
CONST 4	4	[5, 6 , L1]	[]	0
PUSH	4	[4, 5, 6, L1]	[]	0
ACC 3	L1	[4, 5, 6, L1]	[]	0
APPLY3	L1	[4, 5, 6, pc*, [], 0, L1]	[]	2
GRAB 2	L1	[4, 5, 6, pc, [], 0, L1]	[]	0
ACC 2	6	[4, 5, 6, pc, [], 0, L1]	[]	0
PUSH	6	[6, 4, 5, 6, pc, [], 0, L1]	[]	0
ACC 2	5	[6, 4, 5, 6, pc, [], 0, L1]	[]	0
PUSH	5	[5, 6, 4, 5, 6, pc, [], 0, L1]	[]	0
ACC 2	4	[5, 6, 4, 5, 6, pc, [], 0, L1]	[]	0
ADDINT	9	[6, 4, 5, 6, pc, [], 0, L1]	[]	0
ADDINT	15	[4, 5, 6, pc, [], 0, L1]	[]	0
RETURN 3	15	[L1]	[]	0

Table 2: Evolution of the VM registers over the program execution
*pc represents the program counter for the next instruction in the caller.

2.2. Optimizations and Data Representation

To offer a better control over the RAM footprint of their program, we allow the programmer to set up the size of words manipulated by the OMicroB virtual machine independently of the size of words of the hardware architecture. Some architectures we target handle 8-bit words (like PIC18 for example) but we consider not feasible to use an OCaml virtual machine with words shorter than 16-bits, in particular to implement code addresses, block addresses or floats. The size of the "virtual words" is configurable at compile time and the OMicroB virtual machine supports three dedicated representation of data for 16-bit, 32-bit and 64-bit words. The representation of values in OMicroB is uniform as in OCaml. Each value uses a word: either as an immediate value whose size fits in a word, or as an allocated value whose address also fits in a word. An allocated value uses a header (a word) followed by an array of values (words). Values that are allocated in the heap are said to be *boxed*, while immediate values are *unboxed*.

In order to be compatible with the very limited resources of microcontrollers, OMicroB implements various optimizations dedicated to reducing the memory footprint of programs. Notably, we use a particular representation of floating-point values, which differs from the standard implementation of the OCaml VM. In this standard implementation, integers and pointers to the heap

7

are unboxed[3], while floating-point values (and other objects) are boxed. As this indirection can be costly for applications dealing with many floating-point values (for example when reading from analog sensors), we tweak this representation by using *NaN-boxing* of pointers: in *OMicroB* integer values have their least significant bit set to one, floats are unboxed immediate values[4] and pointers to the heap are encoded inside unused NaN values[15]. This representation does limit the address space to 2^{22} different addresses for the 32 bits configuration of OMicroB, but we consider that this is sufficient for the majority of microcontrollers which seldom have more than 4MB of RAM. A single value is kept to represent all NaN values. This representation of OCaml values is described in Fig. 3 and is used by the garbage collector and the standard polymorphic `compare` function to differentiate immediate values (floats[5] and integers) from pointers to heap-allocated values. Configured in 64-bit mode, the virtual machine provides a similar representation. In 16-bit mode, it provides 15-bit floats as 15-bit integers allowing addressing up to 64 kB memory.
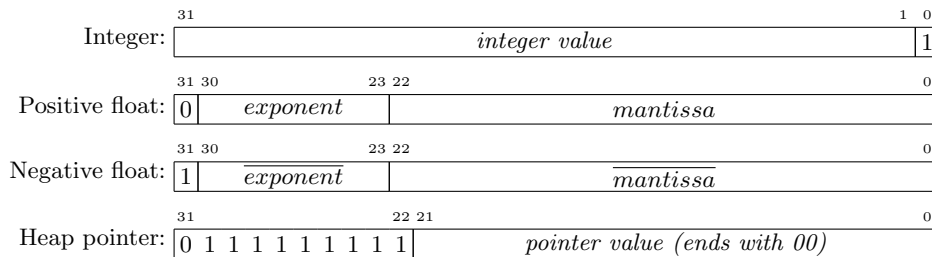
	31			1 0
Integer:		*integer value*		1

	31 30	23 22		0
Positive float:	0	*exponent*	*mantissa*	

	31 30	23 22		0
Negative float:	1	$\overline{exponent}$	$\overline{mantissa}$	

	31	22 21	0
Heap pointer:	0 1 1 1 1 1 1 1 1 1	*pointer value (ends with 00)*	

Figure 3: Representation of OMicroB values (32 bits version)

Another approach for reducing the memory footprint of a program consists in "compressing" it by encoding the bytecode in a C array of bytes (since there are only 149 different OCaml opcodes) where each instruction can be, if needed, followed either by a 1, 2, or 4-bytes constant depending on what is needed to hold the value of the instruction argument. In order to deal with these different sizes of arguments, the bytecode instruction set has been extended with new specialized versions of the bytecode instruction (for example, the `BRANCH` instruction is split in three versions: `BRANCH_1B`, `BRANCH_2B`, and `BRANCH_4B`).

Finally, we run partial evaluation at compile time up to the first input/output (I/O) of the program. The resulting state of the stack and the mutable part of the heap just before the first I/O are then dumped in the microcontroller Flash memory and copied into RAM at starting time. The non-mutable part of the heap (typically made of closures, constant strings and some other constant blocks) is dumped into a "Flash heap segment" never copied into the RAM memory. This almost removes the memory impact of the use of a functional programming language that usually takes up the microcontroller RAM memory with closures. It also speeds up starting time removing the need to compute initialization code on the microcontroller that often need a significant stack size. At last, by expanding toplevel functors at compile time, it improves our algorithm for dead code elimination.

---

[3]As an effect of alignment all addresses end with 0, so integers are represented with a least significant bit set to 1.

[4]Note that typing prevents a floating-point value ending with 1 from being confused with an integer value.

[5]Negative floats have their exponent and mantissa inverted in order to be compatible with `compare`, which at runtime cannot distinguish between two integers or two floats: this way their ordering relation stays the same.

2.3. Garbage Collection

As introduced in the previous section, we allocate parts of the values in Flash memory (drawn in blue in Fig. 4). Flash allocated values are physically immutable but can point to mutable blocks stored in RAM. However, our GC algorithms may move blocks. The RAM heap has then to be divided into dynamic and static heaps. Static blocks (ie. blocks stored in the static heap), should not be moved but their internal pointers can. They are never cleaned by the GC but it does not matter since they are pointed by Flash blocks that are immortal. Pointers from Flash blocks to dynamic blocks (like the red one in Fig. 4) are forbidden but never happen since Flash blocks are immutable.

Figure 4: Representation of OMicroB memories

To distinguish these different pointers and not to try to move the pointers of the Flash memory or the static area, we use two extra bits. As a complement to Fig. 3, here is the representation for 32 bits architectures:

Heap (D) (RAM):

31											20	19																						0
0	1	1	1	1	1	1	1	1	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0

Heap (S) (RAM):

31											20	19																						0
0	1	1	1	1	1	1	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0

Heap (F) (Flash):

31											20	19																						0
0	1	1	1	1	1	1	1	1	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0

Hardware abstraction is also provided by the use of automatic memory management algorithms, with two different garbage collector algorithms from which the developer can choose, one focusing on speed and the other on memory.

The first one is a classical "stop-and-copy" algorithm as described and compared in [16]. Although simple and time efficient, this algorithm needs to use half of the available RAM to perform copies. This is not ideal for microcontrollers, where RAM is often scarce, but its speed is useful for some applications.

9

OMicroB therefore also features a new "mark-and-compact" algorithm that allows using the whole heap memory to store living blocks. Like "stop-and-copy", this GC algorithm always keeps blocks contiguous, allowing fast allocation. When full, the heap is compacted by moving all living blocks to the beginning of the heap. To avoid using extra memory space (except one word by block header as usual), as is the case in the OCaml distribution, this algorithm uses the "pointer-inversion" technique. It performs three passes over the heap:

1. reverse pointers to each living block by chaining memory cells that point to it and put the head of this chain in its header, saving its header in the tail of the chain;
2. plan the destination address of each living block after compaction, then go through the chain of cells that should point to it to update pointers with its upcoming address and restore its header;
3. compact heap by moving living blocks to their planned addresses.

Since the second and third passes jump from block to block over the whole heap, not only over living blocks, this "mark-and-compact" algorithm is slower than "stop-and-copy" on most programs. However, "mark-and-compact" makes better usage of the memory and therefore runs less frequently. Since its execution time depends on the topology of the memory graph, it is in general impossible to advise the optimal choice to the programmer.

2.4. Interrupts and Callbacks

MCU programming makes significant use of hardware interrupts. It is possible to trigger a given task periodically (through timer interrupts), or when the state of an IO changes. We want to be able to give access to this mechanism at the user-level, in the OCaml programming language.

We first need to be able to treat interrupts at the C level. This treatment differs depending on the MCU architecture. For instance, on the AVR architecture, this can easily be done by defining an interrupt handler function through a macro named ISR.

We also need to allow users to define OCaml callbacks which are OCaml functions that can be called from C code. In the standard OCaml virtual machine, callbacks are run by launching a new instance of the virtual machine, solely to run the callback function. This means initializing a new stack and more registers for this new VM instance. We choose a more parsimonious approach by saving the current registers of the VM, and making a function call (using the APPLY instruction). Using this mechanism on interrupts can be problematic. As the interrupt can be fired at any time, including during a GC run or in the middle of the execution of an instruction, saving and changing the VM registers may lead to unpredictable behavior. Our solution is to wait for the end of the current VM instruction to execute the callback. When triggered, the interrup raises a flag which is checked by the VM. Thanks to these mechanisms, the user can define arbitrary callbacks to be called on interrupt. For instance, the code below flashes a LED for 0.5s, every 3s.

```
pin_mode PIN13 OUTPUT;
Timer0.set_period 3000;
Timer1.set_period 500;
Timer0.set_callback (fun () -> digital_write PIN13 HIGH);
Timer1.set_callback (fun () -> digital_write PIN13 LOW);
```

2.5. Compilation Chain

The compilation chain of an OCaml program using OMicroB is shown in Fig. 5. The source code of OCaml programs (with a specific standard library for microcontrollers) is firstly given to

the standard `ocamlc` bytecode compiler. Note that since OMicroB uses this standard compiler, regular OCaml debugging tools such as `ocamldebug` are also entirely compatible with OMicroB.
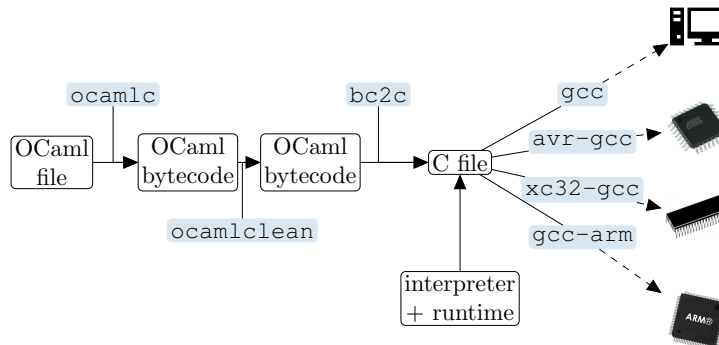

Figure 5: The OMicroB compilation chain

The resulting OCaml bytecode file is then given to the pre-existing `ocamlclean` tool that performs dead-code elimination. Then, a tool specific to this project, called `bc2c`, embeds this bytecode into a C file as multiple arrays of bytes representing the program instructions and global variables for pre-initialized heaps, stack and global variables. This C file is then linked with the interpreter and the runtime library (written in C), and used with a suitable C compiler (such as `avr-gcc` for AVR devices, `xc8` for PIC devices, `gcc` to run the program on a PC for simulation, or some other specific compilation tools for other microcontrollers).

Thanks to the high level of abstraction of the OCaml virtual machine, the structure of OCaml bytecode files is quite stable, only some instructions have been added during the last decade. The main recent changes concern the memory representation of exceptions and closures. To maintain compatibility with OCaml versions, we separate parsing of bytecode files in a generic library called `OByteLib` that exports them into normalized data structures. This library is used by the `bc2c` converter.

2.6. Simulation and Debugging

It should be noted that the execution environment is constrained: no operating system, no memory access check, no simple standard communication interface (keyboard/screen). We therefore make it possible to test the program before transferring it to the microcontroller. Since the VM and its runtime is written in standard C, OMicroB can even be compiled and run on a standard PC for simulation and debugging purposes. This allows for low level debugging such as bytecode execution tracing, checking the memory accesses (using tools such as `valgrind` and `gdb`) and examining the memory dumps.

We also provide a simulator for AVR microcontrollers that can display on a Graphical User Interface (GUI) the state of the General Purpose Input/Output (GPIO) pins of the microcontroller during the execution, as well as its interactions with various analog and digital external devices (LEDs, buttons, displays, . . .).

The following listing presents the language used to describe circuits for use in the simulator. Here, we describe a simple timer, with buttons to increase and decrease the desired time, and a button to start and stop the timer. The graphical interface for this circuit is presented in Fig. 6 on the left. The user can interact with the buttons, as he would on a real circuit. The simulator also displays the current state of the microcontroller, showing on and off pins.

11

```
window width=500 height=260 title="Timer"
lcd16x2 x=30 y=100 e=PIN12 rs=PIN11 d4=PIN2 d5=PIN3 d6=PIN4 d7=PIN5
button x=85 y=40 width=60 color=black label="START" pin=PIN6
button x=405 y=55 width=20 height=20 label="+" pin=PIN7
button x=405 y=25 width=20 height=20 label="-" pin=PIN8
led x=145 y=40 pin=PIN9 color=green
```
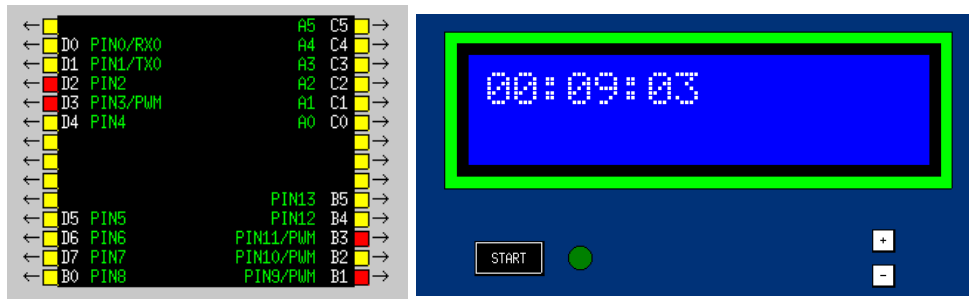

Figure 6: Simulation of a simple timer

This gives developers an easier way to debug their programs, without having to test it directly on an actual microcontroller, which is time consuming and can sometimes cause hardware destruction. The simulator is generic, and supports every AVR-based board supported by OMicroB (Arduino Uno, Arduino Mega, Arduboy). We are currently working on extending it to other architectures supported by OMicroB (the ARM based micro:bit, and PIC32).

2.7. Performance

In this section, we measure the performance of OMicroB on both PC and the BBC micro:bit microcontroller. We run OMicroB with the default command line options, which means using the Mark&Compact GC and activating all optimizations. The test programs are chosen to test different features of the VM and are meant to get an intuition of its performance. They execute function application using lambdas (`church`), recursive functions (`fibonacci`), functions with memory allocation (`takeuchi`), floating-point number calculations (`integr`), object-oriented programming (`object`), and more complex algorithms like sorting using binary search trees (`treesort`) or solving the n-queens problem using linked lists (`nqueens`).

We first compare, in Fig.7 the running time of the test programs using OMicroB and the stock OCaml virtual machine *ocamlrun*. OMicroB is around three times slower than ocamlrun; this means that the choices we had to make to port the OMicroB VM to low-resource architectures do incur performance loss, but not severely so.

We also compare our examples with equivalent C programs compiled with `gcc -O2`. We omit the `church` test (which applies functions to represent Church encoding of integers) from these tests, since this program requires creation of closures, which makes it impractical to implement in C. We also omit the `object` test in C, since it is about handling objects, which are not available in the language. Unsurprisingly, the OCaml programs are generally slower. The cases where OCaml can approach or even outspeed C are the programs with heavy dynamic allocation (`takeuchi` and `treesort`).

Name	OMicroB	ocamlrun	gcc -O2	Python3
church	0.37 s	0.13 s	N/A	1.3 s
fibonacci	0.49 s	0.13 s	0.01 s	0.71 s
takeuchi	0.06 s	0.01 s	0.14 s	0.11 s
integr	5.01 s	2.33 s	0.4 s	6.48 s
treesort	1.47 s	0.48 s	0.15 s	0.36 s
nqueens	4.27 s	1.25 s	0.68 s	11.54 s
object	0.29 s	0.11 s	N/A	0.34s

Figure 7: Performance of OMicroB (32 bits) on PC with Intel® i7-10610U, 16Go RAM

We now compare OMicroB performance to C and MicroPython on the micro:bit device (16MHz, 16 kB RAM, 256 kB Flash memory). We use the subset of OCaml test programs which could be ported to Python without changing too much the structure of the code (the tests that made too much use of recursivity weren't ported to Python as MicroPython's maximal recursion depth is very limited on this device - less than 10). As shown in Figure 8, OMicroB tends to be faster than (Micro)Python, and unsurprisingly slower than C.

We also compare the size in flash memory for OCaml and C programs. There is a small overhead in the flash size of OCaml programs due to the inclusion of the virtual machine. However, this overhead would be compensated for larger source programs.

Name	Run Time			Program Size	
	OMicroB	μPython	arm-gcc -O2	OMicroB	arm-gcc -O2
church	1 302 s	5 900 s	N/A	13 kB	N/A
fibonacci	1 482 s	8 250 s	21 s	12 kB	10 kB
takeuchi	2 241 s	12 120 s	564 s	13 kB	11 kB
integr	22 282 s	37 845 s	6 332 s	16 kB	9 kB
object	1 282s	2 633 s	N/A	25 kB	N/A

Figure 8: Performance of OMicroB (32 bits) on micro:bit

2.8. Application: A Snake Game

OMicroB is lightweight enough to use OCaml for programming microcontrollers with very scarce resources. For example, we have been successful in running an OCaml implementation of the "Snake" game on a little device called Arduboy. This credit card-sized handheld device, used by hobbyists for designing small gaming applications (typically in C), notably features an ATmega32u4 microcontroller with only 2.5 kB of RAM and 32 kB of Flash memory.

The behavior of the game has been described in OCaml using high-level constructs provided by the language, such as exceptions. For example, the following function which defines the main loop of the game uses exceptions to detect (using the `try`/`with` construct) when the game is won or lost:

```ocaml
let play () =
  let max_len = 40 in
  let init_x   = 0 in
  let init_y   = 0 in
  let init_len = 10 in
  let init_dir = South in
  let snake = create max_len init_x init_y init_len init_dir in
  let apple = ref (create_apple snake) in
  draw_first_apple !apple;
  try
    while true do
      move apple snake;
      update_direction snake;
      (* pause for a duration that depends on the current length *)
      delay (max_len - snake_length snake);
    done
  with
  | Win  -> (digital_write Arduboy.g LOW) (* light up the green LED *)
  | Lose -> (digital_write Arduboy.r LOW) (* light up the red LED *)
```

Besides the game logic itself, all hardware interactions (especially with an OLED display using a Serial Peripheral Interface (SPI)) have also been implemented in OCaml. This provides the flexibility and safety of a high-level language for programming low-level interfaces. For example, the booting sequence of the SPI connection is done using the `Array.iter` function to iterate on an array representing the boot program and apply the function `Spi.transfer` on each element of this array:

```ocaml
let boot_program =
  [|
    0xD5 ; 0xF0 ; (* Set display clock divisor = 0xF0 *)
    0x8D ; 0x14; (* Enable charge Pump *)
    0xA1 ; (* Set segment re-map *)
    0xC8 ; (* Set COM Output scan direction *)
    0x81; 0xCF; (* Set contrast = 0xCF *)
    0xD9; 0xF1; (* Set precharge = 0xF1 *)
    0xAF; (* Display ON *)
    0x20; 0x00 (* Set display mode = horizontal addressing mode *)
  |]

let transfer_program prog =
  Array.iter Spi.transfer prog

let boot =
  (* ... omitted for brevity ... *)
  transfer_program boot_program;
```

This program is built using the 16bits version of the OMicroB VM, and just setting a stack size of 64 values and a heap-size of 300 values is sufficient to run it. A picture of the device running the Snake game, together with information about its memory use, is given on Fig. 9. The game logic takes 182 lines of OCaml code, while 90 lines of OCaml code are responsible for the interactions with the display (including the booting sequence displayed above) and just 42 lines are used to represent the rest of the device (buttons, LEDs).

Size on flash	Size on RAM
15,174 bytes	1,060 bytes

Figure 9: An OCaml game of Snake running on an Arduboy

3. Abstracting Architectures and Electronic Components

As our virtual machine was designed in a generic way, it is fitting to provide the end user with a generic, architecture-agnostic way to interact with the microcontroller. We take advantage of OCaml module system, and rely on various features of the language which leverage its strong static typing (such as Algebraic Data Types, polymorphic variants, and Generalized Algebraic Data Type), to do so.

3.1. Hardware and Foreign Function Interface

To define low-level functions that manipulate the hardware, we can use the Foreign Function Interface (FFI) of OCaml, which allows the user to call functions written in C. We use this functionality to give access to MCU primitives, such as the one that manipulate hardware directly. Among these may be the functions to read and write digital or analog signals to the GPIO, or to control the timers of the MCU. Of course, we want to use the high-level features of OCaml to somewhat abstract and protect these low-level calls.

3.2. Algebraic Data Types and pattern-matching

As we want to provide a generic description of the hardware, we may start with the GPIO interface of the microcontroller. It is constituted of a finite set of pins, on which the program may read or write digital or sometimes analog signals. In a typical OCaml program, we may implement these pins with an Algebraic Data Type (ADT). The listing below shows such an ADT for an hypothetical microcontroller with three `pins`. Note that the last constructor, `PINAn` carries its `pin` number as a value of type `int`: ADTs may be simple enumerated types, or more general sum types with data specific to each constructor.

If we wanted to write a function that returns the `pin` number, we could leverage the pattern-matching feature of the OCaml language, by defining a different behavior for each constructor.

```
let pin_number (p : pin) : int =
  match pin with
  | PIN0 -> 0
  | PIN1 -> 1
  | PINAn n -> n
```

15

3.3. Generalized Algebraic Data Types

A powerful extension of these types are Generalized Algebraic Data Type (GADT)s. They allow to specify more constraints on type parameters of value constructors, and to use existentially quantified type variables. In a GADT, the type declaration is parameterized by zero, one or several type variables. Each constructor may use a different instantiation of these type variables. In the example below, we use this feature to differentiate between `pins` that do or do not support reading analog signals. Among other features, this construction allows us to define functions and patterns matching on a subset of the constructors of the type.

```ocaml
type noanalog
type analog
type 'a pin =
  | PIN0  : noanalog pin
  | PIN1  : noanalog pin
  | PINA0 : analog pin

let analog_write (p : analog pin) (lvl : int) =
  match p with
  | PINA0 -> unsafe_analog_write 0 lvl
```

Take for instance the `analog_write` function above. Its primary role is to protect calls to `unsafe_analog_write`, an external C function defined using the FFI. It takes as input the number of an analog pin, and an integer to write to this pin. With the definition below, the pattern-matching of `analog_write` is total, because, by typing, we know that parameter p,which has type `analog pin`, can only be `PINA0` , since this is the only constructor of this type. Conversely, the type checker would reject calls to `analog_write PIN0` or `analog_write PIN1`.

3.4. GADTs and polymorphic variants

The definition above is a good step in the direction of a precise `pin` type, but we wish to refine it once more. Indeed, on a microcontroller, there may be some `pins` that support both Pulse-Width-Modulation (PWM) writing and analog reading, some that support one but not the other, some that support neither, and some that support other features. To make our specification generic, we need a way to specify some kind of "list" of the features of a given `pin`. This may be achieved using OCaml's polymorphic variants [17]. These constructs allows one to use variant "tags" that are not tied to a specific declared type. Such a tag is noted with a backtick (`` `DREAD ``), and a variant type can be given with a pipe-separated list of tags, enclosed with square brackets. For instance, the type `[`DREAD | `DWRITE]` has two tags. Polymorphic variants allow for a limited form of subtyping. The polymorphic variant type `[< `DREAD | `DWRITE ]` (note the `<`) denotes a sub-type of any type that contains at most the `` `DREAD `` and `` `DWRITE `` constructors. In particular, it is a subtype of both `[ `DREAD]` and `[`DWRITE ]`. We can use this mechanism to specify `pins` that have different capabilities. Suppose for example that all pins support digital reading and writing, that only `PIN1` supports PWM writing, and that only `PINA0` supports analog reading, we would write the following type declaration.

```ocaml
type 'a pin =
  | PIN0 : [< `DREAD | `DWRITE ] pin
  | PIN1 : [< `DWRITE | `DREAD | `PWM ] pin
  | PINA0 : [< `DWRITE | `DREAD | `AREAD ] pin
```

16

From here, we can give a type to functions that is specific to pins with a given feature. For instance, the call `digital_read` on `[ 'DREAD ] pin` type-checks, while the call to `analog_read` function type-checks only for a `[ 'AREAD ] pin`. Therefore, we can pass `PINA0` to the `analog_read` function, but not `PIN0` or `PIN1`; this is enforced entirely by the type-checker, at compile time.

3.5. A Common Module Interface

Now that we have defined the type of pins using a GADT, and that we know how to manipulate the hardware, we can outline a minimal list of core features we wanted to support for all devices. Most of the architectures require a pin to be set either as an input or an output. This configuration can also be changed at runtime. We therefore provide a type `mode` as well as a function `pin_mode`, which accepts any `pin`. Once the mode is set, the user can either read or write a digital value (represented as a binary type `level`) on the pin with functions `digital_read` and `digital_write`. Moreover, most devices also allow some of the GPIO pins to process analog signals; we provide two functions `analog_read` and `analog_write` to treat these signals. We represent analog values by integers in the range $[0; 1024[$. Finally, we provide two primitives to handle time: `delay` pauses the execution for a given number of milliseconds, while `millis` returns the number of milliseconds elapsed since the start of the program. These names are inspired from the Arduino library. Our goal was indeed to provide an easy transition from the Arduino environment to OMicroB.

We package these definitions in a single module type. In OCaml, modules allows the programmer to define a reusable set of types and functions. The implementation of modules may be abstracted under an interface (or signature), using the **module type** syntax. This is what we do below.

```
module type MCUConnection = sig
  type 'a pin = ...
  type mode = INPUT | OUTPUT
  type level = LOW | HIGH
  val pin_mode : 'a pin -> mode -> unit
  val digital_read : [ 'DWRITE ] pin -> level
  val digital_write : [ 'DREAD ] pin -> level -> unit
  val analog_read : [ 'AREAD ] pin -> int
  val analog_write : [ 'PWM ] pin -> int -> unit
  val delay : int -> unit
  val millis : unit -> int
end
```

These functions are defined as calls through the FFI to device-specific C primitives. The OCaml library is organized in modules for each of the supported architecture, each with an implementation of the `MCUConnection` signature. In the case of architecture's having several controllers (e.g. Avr), a root module contains architecture-specific definitions while the device-specific ones (mostly the `pin` type) are defined in sub-modules.

Having introduced a consistent naming scheme for our types and functions means that the end user's code can target interchangeably any architecture, simply by changing the opened module (and possibly sub-module). This is done automatically by OMicroB when choosing a target device, using the `-open` option of `ocamlc`. This means that a program such as the one below (which blinks a LED connected on `PIN2`) can be compiled for any microcontroller (with at least two pins) by simply changing a command line option.

```
pin_mode PIN2 OUTPUT;
while true do
  digital_write PIN2 HIGH; delay 500;
  digital_write PIN2  LOW; delay 500
done
```

3.6. Describing High-Level Components with Functors

Having defined a sufficient set of primitives, we can move onto a higher level of abstraction, by representing not only connections, but electronic components, such as LEDs, push buttons, or more complex devices like LCDs. We rely again on OCaml's module system. We make use of the module interface `MCUConnection` described in 3.5. This interface is implemented once per device. It provides the basic features to interact with the microcontroller.

To program external devices, we must write code that in some way interacts with this interface. In OCaml, this can be done using functors, that is, functions that take modules as input and build new modules. For instance, the functor `MakeLCD` below expects a module that respects the `LCDConnection` interface. This interface can be instantiated by extending the `MCUConnection` interface with a number of pins through which the display communicates with the MCU. The instantiated functor returns a module of type `Display`, that exposes high-level functions that manipulate the component. The definition of `MakeLCD`, not shown here, uses the lower-level functions from `MCUConnection` to implement these.

```
module type Display = sig
  val init: unit -> unit
  val print_string: string -> unit
  val set_pixel: int -> int -> bool -> unit
  val clear_screen: unit -> unit
end
module type LCDConnection = sig
  type 'a pin
  type level
  include Circuits.MCUConnection
    with type 'a pin := 'a pin with type level := level
  val rsPin: [ `DWRITE ] pin
  val enablePin: [ `DWRITE ] pin
  val d4Pin: [ `DWRITE ] pin
  val d5Pin: [ `DWRITE ] pin
  val d6Pin: [ `DWRITE ] pin
  val d7Pin: [ `DWRITE ] pin
end
module MakeLCD(L: LCDConnection): Display
```

Using this approach, we can also provide two interfaces for communication protocols I^{2}C and SPI. [18]. These interfaces are implemented for every model of microcontroller which supports them, and can be used to instantiate higher level functors (for instance, for the SSD1306 display, which uses the I^{2}C standard).

3.7. A Generic Application: The MicroPong Game

We now illustrate the usage of this module system, from the user point of view. We built a multiplayer version of the well-known game "Pong". This is an opportunity to demonstrate how the communication between several microcontrollers can be set up using only OCaml code. Two

of these microcontrollers are designated as players, using buttons to move their respective paddles. The last microcontroller acts as the server, receiving the players input and displaying the game on a connected screen. The players also display their current score on their respective screens.
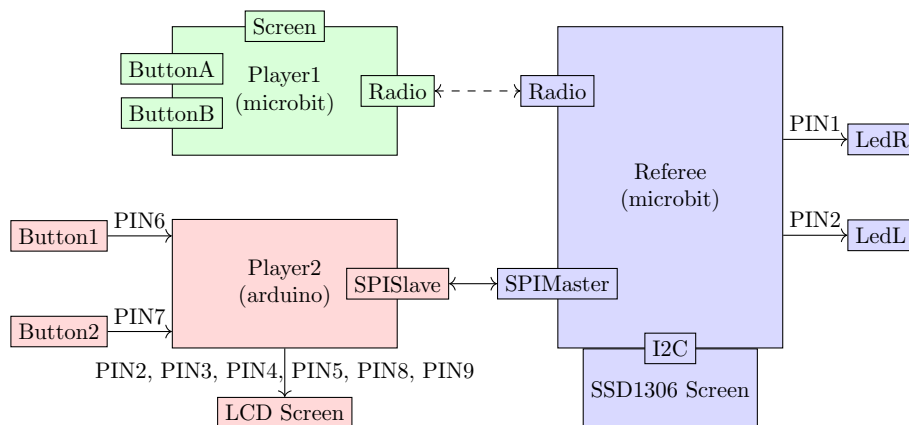
3.7.1. Devices Used

Figure 10: MicroPong app structure

In this application, we will use the BBC micro:bit, an ARM-based education-focused microcontroller, supported by OMicroB. The standout components of the micro:bit are a 5x5 LED-matrix display, two mounted-on buttons, an accelerometer and a Bluetooth/radio antenna. The ARM Cortex-M0 MCU on the micro:bit provides 16 kB of RAM, which is more than most AVR-based platforms, and useful for more memory-intensive applications.

We have chosen to use two BBC micro:bit for both the server and one of the player, so that they can communicate using the radio module. To experiment with our approach, we make the second player communicate with the server not via radio but via SPI; this means both players are running slightly different progams. SPI is a wired, synchronous communication protocol implemented by most microcontrollers, including the BBC micro:bit. Thanks to the genericity of our approach, we may use any microcontroller that supports SPI as the second player without changing its program; in our experiment, we chose the Arduino Uno. We show in Fig. 10 how the three microcontrollers are interfaced. In this figure, the plain arrows denote a physical connection between two elements, while a dashed arrow denotes a wireless (radio) connection.

3.7.2. Program

We show below how the server's modules are instantiated. Notably, the instantiation of the I2C module requires an `address` parameter. The I2C module can then be used to instantiate a `MakeSSD1306` functor, which is used to control a `SSD1306` OLED display.

```
module I2C = I2C(struct let address = 0x3C end)
module Scr = Ssd1306.MakeSSD1306(I2C)
module%comp LeftLed = MakeLed(struct let cPin = PIN1 end)
module%comp RightLed = MakeLed(struct let cPin = PIN2 end)
module%comp SPIM = MakeSPIMaster(struct let sPin = PIN0 end)
```

19

As seen in this example, we have defined a syntactic extension for instantiating modules, using the OCaml PPX rewriter system. Indeed, in our implementation instantiating a module can be verbose, as it requires the user to include the `MCUConnection` module along with the actually useful declarations. We hide this behind the **`module%comp`** which indicates the start of an extension point. For instance, the third line of the listing expands to the module declaration below.

```
module LeftLed = MakeLed(struct
    include MCUConnection
    let cPin = PIN1
end)
```

The rest of the server's code (80 lines of OCaml) uses these modules generic interfaces, but does not rely on any other device-specific code. We give below a (simplified) snippet of the communication code for the server. The server communicates the player's scores over radio. It then receives the input of the Left player over SPI, and the one of the Right player over radio. Note that this piece of code does not depend on any architecture-specific functions, but only on the instantiated modules. This is also the case for the rest of the server code, which is mainly concerned with calculating the trajectory of the ball and displaying it, using the `SSD1306` display module. There is no dependency between the logic of the program and the chosen device.

```
Radio.send ("sr" ^ string_of_int !scoreR);
Radio.send ("sl" ^ string_of_int !scoreL);
let lpos = int_of_char (SPIM.transmit !scoreL)
and rpos = Radio.recv () in
  if lpos <> 0 then lY := lpos;
  if String.length rpos = 1 then rY := rpos.[0];
```

4. Abstracting Concurrency: The OCaLustre Extension

As we have seen in the previous section, the PPX language extension system available in the standard OCaml compiler can give developers powerful means of providing tailor-suited programming constructs for specific uses. Since programs for microcontrollers are inherenlty concurrent because of their fast and often unordered reactions to external stimuli, we thus propose another use of PPX to define a specialized concurrent programming extension to the OCaml language that suits the programming of embedded software.

This extension, called OCaLustre (based on the prototype presented in [19] and extended with a multi-clock system and a new sequential code generation), is based on the synchronous dataflow language Lustre [12]. The OCaLustre extension follows the main principle of Lustre, where the time taken for computing the outputs values of a program based on its inputs values is considered as being nil. This principle, known as the *synchronous hypothesis* [11], results in a powerful abstraction that provides a deterministic model of concurrency. Indeed, scheduling of all the (possibly interdependent) concurrent logical aspects of a program is computed by the compiler, thus producing sequential code of an imperative language that is semantically equivalent to the given synchronous program. Note that phenomena of *causality loops*, where two concurrent elements of a program depend instantaneously on each other, are detected by the compiler, which rejects such semantically undefined programs. OCaLustre produces imperative OCaml code, entirely compatible with OMicroB.

20

4.1. Synchronous Nodes and Operators

As in Lustre, the main construct of an OCaLustre program is called a *node*, which is akin to an *instantaneous* function computing values for output flows from values of input flows. The following code defines a node (with the **let%node** notation), called `plus_minus`, which receives two `a` and `b` flows as inputs, and produces two `p` and `m` flows which values are respectively the sum and the difference of the inputs:

```
let%node plus_minus (a, b) ~return:(p, m) =
  p = a + b;
  m = a - b
```

The body of a node is a system of equations that defines the values of its outputs (and possible local flows). It is solved at each program's "tick" (called a synchronous *instant*), in a logical zero time. The `p` and `m` flows are thus considered as two *concurrently* computed outputs.

Besides classical arithmetic/boolean operators, OCaLustre provides various synchronous operators. The `->>` operator (akin to the Lustre *followed-by* - or `fby` - operator) permits the definition of a flow as a constant value for the first instant of the program, followed by the *previous* value of an expression for the subsequent instants. For example, the following node defines a `cpt` flow that is 0 at the beginning of the program, and then is the previous value of `cpt + 1` for every subsequent instant, representing an incrementing counter. This flow is reset (by a modulo operation) each time the counter is greater than the value of the `reset` input flow:

```
let%node count (reset) ~return:(cpt) =
  cpt = (0 ->> (cpt + 1)) mod reset
```

The PPX rewriter system transforms a valid (possibly annotated) OCaml AST into another valid OCaml AST. OCaml's system of annotations is thus used in OCaLustre to define positive and negative *sampling* operators, that provide a way to condition the presence of flows using other boolean flows. We use the [**@when** b] (resp. [**@whennot** b]) annotation to convey the fact that a flow has a value *only* when the `b` flow evaluates to *true* (resp. *false*), and in other cases is considered as having no value (and thus should not be evaluated): we say that this flow is positively (resp. negatively) sampled by `b`. For example, the following node counts the number of times its inputs are true together (modulo 100) because `count` cannot be called with an absent input:

```
let%node call_count (x, y) ~return:(clk, z) =
  clk = (x && y);
  z = count (100 [@when clk])
```

The `clk` flow is called the *clock* of `z`, which is the flow which determines the presence of `z`. Flows that are not explicitly sampled are said to be on the *base* clock of their node: they are computed each time the node in which they are defined is applied.

Figure 11 shows the possible evolution of the values of the different flows of the `call_count` node over time. A blank cell indicates that a flow is *absent* (i.e. it does not have a value) for the corresponding instant.

A **merge** operator is used to combine oppositely sampled flows: its first parameter is a clock (a boolean flow), its second parameter is a flow positively sampled by this clock ([**@when** clk]) and its third parameter is a flow negatively sampled by the same clock ([**@whennot** clk]). For example, the following code combines the value of two flows oppositely sampled. Since the clock that samples

21

instant	0	1	2	3	4	5	6	...
x	true	true	true	false	true	true	false	...
y	true	false	true	true	true	true	true	...
clk	true	false	true	false	true	true	false	...
z	0		1		2	3		...

Figure 11: Clock sampling

them switches between being true and false at each instant, then the output flow itself flip-flops between the value 23 and value 42.

```
let%node flip_flop () ~return:(res) =
   clk = false ->> (true ->> clk);
   x = 23 [@when clk];
   y = 42 [@whennot clk];
   res = merge clk x y;
```

Figure 12 shows the evolution of the values of the differents flows of the `flip_flop` node over time.

instant	0	1	2	3	4	5	6	...
clk	false	true	false	true	false	true	false	...
x		23		23		23		...
y	42		42		42		42	...
res	42	23	42	23	42	23	42	...

Figure 12: Clock sampling and merge operator

The **merge** operator is the only OCaLustre operator that works on flows that do not share the same clocks. For example, the following code is incorrect (and the OCaLustre compiler reports the error) because it is attempting to add together two flows on two different clocks:

```
let%node add_wrong (a,b) ~return:(wrong) =
   clk1 = false ->> (true ->> clk1);
   clk2 = true ->> (false ->> clk2);
   x = a [@when clk1];
   y = b [@when clk2];
   wrong = x + y (* error *)
```

Additionally, clocks themselves can be sampled, and various layers of sampling can be achieved, as seen in the following example, where each flow is annotated with a comment identifying its clock:

```
let%node clocks_of_clocks (b, c) ~return:(res) =
   d = b [@when c]; (* c *)
   e = 4 [@whennot c]; (* not(c) *)
   x = 5 [@when d]; (* d *)
   y = 6 [@whennot d]; (* not(d) *)
   z = merge d x y; (* c *)
   res = merge c z u (* base clock *)
```

The intricacies of sampling for this example are displayed in Figure 13. The array represents the

evolution of the values of each flow over time. Each value is represented in the same colour as the flow that samples it (the base clock - for which values are present at every instant - is represented in green).

instant	0	1	2	3	4	5	6	...
b	true	false	true	false	true	true	false	...
c	true	true	false	false	true	false	true	...
d	true	false			true		false	...
e			4	4		4		...
x	5				5			...
y		6					6	...
z	5	6			5		6	...
res	5	6	4	4	5	4	6	...

Figure 13: Multiple clock sampling

Clocking rules can be seen as a type system [20], and OCaLustre uses an inference algorithm to determine the clock of each flow.

4.2. Sequential Code Generation

The OCaLustre compiler generates sequential OCaml code from each node definition that is present in an OCaLustre program, as well as code for the main loop of the program, that each instant reads its inputs, runs the synchronous program, and writes its outputs. The generated code has a low memory footprint, on RAM as well as on Flash memory: for example, the following is the sequential OCaml code produced by compilation of the count node:

```
let count () =
  let _cpt_aux1_fby = ref 0 in
  fun reset ->
    let _cpt_aux1 = !_cpt_aux1_fby in
    let cpt = _cpt_aux1 mod reset in
      _cpt_aux1_fby := (cpt + 1); cpt
```

This code only uses the functional and imperative kernel of the OCaml language, and thus induces a limited sequence of bytecode instructions. The produced bytecode is 20 instructions long, compatible with OMicroB, and can run on microcontrollers with limited resources.

OCaLustre can also be set to produce non-allocating OCaml code, creating functions that take a mutable record value as input, representing the internal state of a node, and update it at each instant of the program. Thus, the generated code never triggers the garbage-collection algorithm, since every update is done in place, and no memory is allocated. Morever, it is possible to analyse the execution time of the program [21]. However, this code tends to result in a longer bytecode for the same OCaml application, and we thus give the choice to the user to decide between timing guarantees and a smaller memory footprint. Moreover, calls to standard OCaml code may themselves still trigger the GC, and this "non-allocating" version is thus only compatible with synchronous programs which do not make use of all the features of OCaml.

4.3. Example: Chocolate Tempering Machine

We now present a concrete application using OCaLustre: a chocolate tempering machine. This machine is useful for melting chocolate while keeping it at a controlled temperature. We prototype such a machine using an arduino uno, a heating component and a temperator sensor.

4.3.1. Circuit

The circuit is presented on Fig. 14. It contains two buttons, controlling the desired temperature, a screen displaying the desired and actual temperature, a heating element, and a temperature sensor. We assume that the heating element simply takes a digital signal as input, indicating if it should be on or off. We also assume that the temperature sensor produces an analog signal between 0 and 5V, that can be read using `analog_read`.
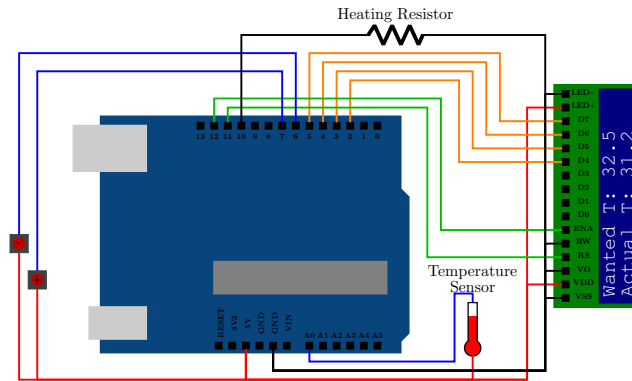
Figure 14: Temperature controlled chocolate heater

4.3.2. Program

We now specify the behavior of the system as an OCaLustre program. The first two nodes, `thermo_on` and `set_wanted_temp` specify the behavior of the buttons. Pressing both buttons at the same time toggles the system on and off. Otherwise, pressing only one of the buttons increases or decreases the desired temperature, expressed in tenth of Celsius degrees.

```
let%node thermo_on(p,m) ~return:(b) =
  b = (true ->> if p && m then not b else b)
let%node set_wanted_temp (p,m) ~return:(w) =
  w = (325 ->> if p then w+5 else if m then w-5 else w)
```

The following node calculates the "heating proportion" (expressed in percent). It depends on the difference between wanted and current temperature, and should not exceed 100. It should also not vary too quickly in time, and is therefore calculated using the proportion at the previous time step.

```
let%node update_prop (wtemp,ctemp) ~return:(prop) =
  delta = min (10,max (-10,wtemp-ctemp));
  delta2 = if delta < 0 then (-delta * delta) else (delta*delta);
  offset = min (10,delta2);
  pre_prop = (0 ->> prop);
  prop = min (100,max (0, (pre_prop+offset)))
```

24

The `timer` node implements software Pulse-Width-Modulation. The signal produced is true for `prop` percents of the time. By averaging this boolean signal overtime, the physical system will behave as if it was fed an analog signal with the corresponding intensity. The `heat` node feeds the calculated property to the `timer` node to determine if the resistor should be on or off. As calculating a new proportion is somewhat costly, it is only done once every 10 steps.

```
let%node timer (prop) ~return:(alarm) =
  time = (0 ->> (time + 10)) mod 100;
  alarm = time < prop;

let%node heat (w,c) ~return:(h) =
  count = (0 ->> count + 1) mod 10;
  update = (count = 0);
  prop = merge update
              (update_prop (w [@when update],c [@when update]))
              ((0 ->> prop) [@whennot update]);
  h = timer (prop)
```

Finally, all these nodes are put together by `thermo`. The outputs are a signal indicating if the system is `on`, the `wanted` and `real` temperatures, and if the `resistor` should turn on.

```
let%node thermo(p,m,realtemp) ~return:(on,wanted,real,resistor) =
  on = thermo_on (p,m);
  wanted = set_wanted_temp (p[@when on], m[@when on]);
  real = realtemp [@when on];
  heat = real < wanted;
  resistor = merge on heat (false [@whennot on])
```

The inputs and outputs of the main node (`thermo`) are passed from and to the environment through OCaml functions that are easily written thanks to the high-level interface provided with OMicroB. When the compiler is given the name of the main node, the generated code automatically calls these interfacing functions at the beginning and end of every instant:

```
let input_thermo () =
  let plus_lvl = digital_read plus in
  let minus_lvl = digital_read minus in
  let plus = bool_of_level plus_lvl in
  let minus = bool_of_level minus_lvl in
  let real_temp = read_temp () in
  (plus,minus,real_temp)

let output_thermo (on,wanted,real,res) =
  digital_write resistor (if res then HIGH else LOW);
  if on then
      print_temp wanted real
  else
    begin
      LiquidCrystal.home lcd; (* move cursor back to the beginning *)
      LiquidCrystal.clear lcd;
      LiquidCrystal.print lcd "..."
    end
```

4.3.3. Simulation

In Fig.15, we show the interactive simulation of the program described above. The behavior of the chocolate is not simulated, so the user has to input manually the "actual" temperature of the simulated chocolate using the analog slider. The LED on the left indicates if the heating is on or off.

Figure 15: Simulation of the heater

5. Discussion and Conclusion

In this article, we have shown how it is possible to use a feature-rich programming language like OCaml in embedded systems, offering to developers safety thanks to its type system and its synchronous extension OCaLustre on the one hand, and on the other hand expressiveness thanks to its multi-paradigm approach (functional, imperative, object, modular). We have been able to run relatively complex applications, even considering the scarcity of memory (only a few kilobytes of RAM) on some microcontrollers. This thanks to numerous optimizations both in terms of code size reduction (dead code detection, specialized instructions on argument size, elimination of useless instructions) and memory usage reduction (partial evaluation, use of Flash memory for non-moveable data, choice of GC algorithms).

Several similar efforts and experiments have been made in the past to port high-level languages for programming microcontrollers. We compare these experiments on the following criteria: ease of implementation, smallness, expressivity, safety, efficiency, and on the language families: scripting languages, dynamic languages, statically typed languages, imperative, object and functional languages, and then for the concurrency with synchronous languages. The ease of implementation of the language is a criterion that has allowed certain ports to be made, such as for imperative languages like Forth with tinyForth[6] which implements the interpreter in AVR assembler, or the more generic PIcForth[7] for PIC16Fxxx family. It is the same for functional programming with Scheme kernels like microScheme [22] which simplifies memory management. Still in Scheme, the PicoBit experiment [3] follows the virtual machine approach. One of the interests of Lisp machines is to be able to integrate also the Read-Eval-Print Loop (REPL) and the compiler to the bytecode, the whole compiler and bytecode interpreter with the REPL remaining small as for example the Ribbit machine [4].

---

[6]In Japanese: `http://middleriver.chagasi.com/electronics/tforth.html`
[7]`https://rfc1149.net/devel/picforth.html`

The Lua scripting language, whose implementation is small, has a version for microcontroller [8] which allows to use the different programming features targeting a virtual machine with memory management. In this category of dynamic languages, easy to program, we also find the Python and JavaScript familiesusing code source interpreter or virtual machine approaches. We can note many experiments in JavaScript as for the most widespread the Espruino [9] interpreter or [10] using a virtual machine. Similarly for Python, microPython [2] uses a virtual machine to run the produced bytecode by its compiler. It is even distributed directly loaded on the Pyboard hardware, a Cortex M4 with 512 kB Flash ROM and 128 kB RAM.

The efficiency of these languages is still low, hence experiments such as Warduino [23] (Web Assembly for Arduino) or WAMR [11] (Web Assembly Micro Runtime), which target Web Assembly to improve the speed of execution. For a greater efficiency of imperative languages in a statically typed framework, there are implementations of Ada [24] and Rust on microcontroler [12]. Memory management is manual, although for Rust it is safer thanks to its type system. On the other hand tinyGo [13] for Goland has an automatic memory management. In the same way, in object oriented languages, experiments in Java and C# have followed the virtual machine approach as Darjelling [1] or microEJ [14] for Java, and .NET Nanoframework [15] even if it means not implementing all the features of these very rich languages.

While these languages offer nice high-level abstractions, and sometimes even some functional programming features, we are more interested in a fully functional model. Closer to our work are PICOBIT [3] and AtomVM [16] which aims to port the Scheme (resp. Erlang) programming language to microcontrollers, using a virtual-machine based approach. Haskell, a statically-typed, purely functional programming language was also used to program microcontrollers through Haskino [25], using a monadic method restricting the programmer to a subset of the language. Juniper [9], an ML-style Functional Reactive language was also designed specifically to be run on Arduino cards. In contrast to these two latter works, we preferred to port the entire existing OCaml language, in order to take advantage of existing OCaml programs and libraries.

For concurrency models, the same criteria of smallness, expressiveness and safety are necessary. For this purpose, the classical models of preemptive threads are not desirable. Lua brings a model of co-routines, quite simple to implement but low level to program. In the languages of the ML family several systems are experienced. RTMLton [26] proposes an SML runtime for real time systems where the modified GC facilitates the predictability of execution times, and the concurrent model is thread based. Closer to our concerns, the SynchronVM virtual machine (previously senseVM [27]) provides a higher-order concurrency model based on message passing, based on synchronous channels (à la Concurrent ML [28]) by integrating hardware interrupts into this mechanism. Handling a mini-Caml, the size of the programs and the execution times are quite large compared to OMicroB, nevertheless the synchronous channels approach is an interesting abstraction for software components.

---

[8] http://www.eluaproject.net
[9] https://github.com/espruino/Espruino
[10] https://jerryscript.net/
[11] https://github.com/bytecodealliance/wasm-micro-runtime.
[12] https://www.rust-lang.org/what/embedded
[13] https://tinygo.org
[14] http://www.microej.com
[15] https://www.nanoframework.net/,
[16] https://github.com/atomvm/AtomVM

We leveraged the low-level abstractions given by OMicroB to provide a way to write portable code for multiple microcontroller architectures, with minimal hurdle for the user. Static typing and automatic memory management increase reliability, but it is also necessary to abstract the circuit composition and the concurrency model to provide guarantees at these levels. OCaml-like typing based on polymorphic variants and GADT, as well as the use of functors, allows to compose more easily the different components of a circuit while guaranteeing the correctness of the communications between them. Of course, it would have been possible to build this kind of abstraction through inheritance and structural sub-typing of OCaml objects, with greater memory consumption. Similar behavior can be achieved with Python's object layer for code reuse via inheritance, but to guarantee static typing, it would be necessary to have a static analysis on the object code in the presence of late binding, which isn't always easy, even though static analyses using abstract interpretation are now focusing on dynamic languages, as for example with Python's MOPSA tool [29]. The OCaml module system allowed us to build a compositional and generic component system. This allows us to represent circuits, and could eventually be abstracted through a Domain Specific Language, or a graphical editor. Finally, the extensibility of the OCaml language allowed us to define a synchronous extension which also uses the OCaml type system to provide strong guarantees about its execution.

Our future works include porting OMicroB to new architectures, which should require minimum efforts given the genericity of our approach. For example Raspberry Pi or ESP microcontrollers, which offer interesting applications for the Internet of Things, but we would also like to broaden the targets towards FPGA reconfigurable circuits as shown by the work on O2B which uses the OMicroB machinery on a softcore processor to then call specialized circuits [30]. Moreover, we would like to try an innovative way to interface the synchronous part (OCaLustre nodes) and the sequential part (the code of the host language – OCaml, in this case) of a program. In the current implementation, external calls from OCaLustre nodes to OCaml functions block synchronous flows, breaking the concept of the synchronous clocks. We are currently designing a new Foreign Function Interface (FFI) for OCaLustre using concurrency thanks to Virtual Machine threads. This new interface would allow to run external calls to sequential functions during multiple synchronous ticks and to smoothly manage them through synchronous flows.

Other synchronous programming languages, inspired by Esterel [31], that do not rely on flows but on *events* may also be well adapted to the programming of embedded systems, as such systems can be seen as interactive systems for which timely reaction to events is paramount. For example, the ReactiveML language [32], based on a synchronous reactive model, seems to be an ideal candidate to develop reactive embedded OCaml-like programs. Some early attempts of adapting ReactiveML for OCaPIC/OMicroB have been conducted, but the large size of the generated OCaml bytecode is a current drawback. Nonetheless, we maintain interest in this approach and are planning on pursuing our work on this subject, as having a model that blends OCaLustre and "microReactiveML" could be very valuable.

OCaml version 5.0 was recently released with several important modifications, including new virtual machine instructions, parallel programming including GC and a relaxed memory model. Some points of GC parallelization could be used to build a real-time GC, or at least one that progresses at each step in a bounded time. However, the need for progression and synchronization increases the size of the runtime library, making it impossible to run OCaml programs in small microcontrollers. For the time being, we recommend keeping the size of OMicroB executables small, to target low-resource microcontrollers with a pre-allocated synchronous part and a computation part that can freeze computation. Nevertheless, the possibility of guaranteeing that the computations launched do not disturb the synchronous part should be studied to extend the programming model.

For more communicating applications requiring system libraries, we are interested in the uniker-nel approach of MirageOS [33] which includes only the components necessary for the application. The idea is to be able to use them directly, because the program and MirageOS components are written in OCaml and easily interoperable, thus allowing to dimension correctly the cloud parts for IoT applications.

Finally, we would like to build a more user-friendly IDE (Integrated Development Environment), reminiscent of the Arduino environment, in order to use OMicroB as an educational tool. For this to gain a wider adoption, it would be useful to be able to run, as MicroPython, Ribbit or WARDuino do, an interactive OCaml REPL on the microcontroller.

Acknowledgments

References

[1] N. Brouwers, P. Corke, K. Langendoen, Darjeeling, a Java Compatible Virtual Machine for Mi-crocontrollers, in: ACM/IFIP/USENIX Middleware '08 Conference Companion, ACM, 2008, pp. 18–23. doi:10.1145/1462735.1462740.

[2] C. Bell, MicroPython for the Internet of Things: A Beginner's Guide to Programming with Python on Microcontrollers, Apress, 2017. doi:10.1007/978-1-4842-3123-4.

[3] V. St-Amour, M. Feeley, PICOBIT: A Compact Scheme System for Microcontrollers, in: 21st International Symposium on Implementation and Application of Functional Languages (IFL 2009), Vol. 6041 of Lecture Notes in Computer Science, Springer, 2009, pp. 1–17. doi:10.1007/978-3-642-16478-1_1.

[4] S. Yvon, M. Feeley, A Small Scheme VM, Compiler, and REPL in 4k, in: Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL 2021, ACM, 2021, pp. 14–24. doi:10.1145/3486606.3486783.

[5] Y. Minsky, OCaml for the Masses, Commun. ACM 54 (11) (2011) 53–58. doi:10.1145/2018396.2018413.

[6] B. Vaugon, P. Wang, E. Chailloux, Programming Microcontrollers in Ocaml: the OCaPIC Project, in: International Symposium on Practical Aspects of Declarative Languages (PADL 2015), Vol. 9131 of Lecture Notes in Computer Science, Springer Verlag, 2015, pp. 132–148. doi:10.1007/978-3-319-19686-2_10.

[7] X. Leroy, The ZINC experiment: an economical implementation of the ML language, Tech. Rep. 117, INRIA (1990).

[8] K. Sawada, T. Watanabe, Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, in: Companion Proceedings of the 15th International Conference on Modularity, MODULARITY Companion 2016, ACM, 2016, p. 36–44. doi:10.1145/2892664.2892670.

[9] C. Helbling, S. Z. Guyer, Juniper: a functional reactive programming language for the Arduino, in: 4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016), ACM, 2016, pp. 8–16. `doi:10.1145/2975980.2975982`.

[10] J. Colaço, B. Pagano, M. Pouzet, SCADE 6: A formal language for embedded critical software development (invited paper), in: Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering (TASE 2017), IEEE Computer Society, 2017, pp. 1–11. `doi:10.1109/TASE.2017.8285623`.

[11] N. Halbwachs, Synchronous Programming of Reactive Systems, in: 10th International Conference on Computer Aided Verification (CAV 98), Vol. 1427 of Lecture Notes in Computer Science, Springer, 1998, pp. 1–16. `doi:10.1007/BFb0028726`.

[12] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, Lustre: A Declarative Language for Programming synchronous systems, in: 14th annual ACM Symposium on Principles of Programming Languages (POPL'87), Association for Computing Machinery, 1987, pp. 178–188. `doi:10.1145/41625.41641`.

[13] S. Varoumas, B. Pesin, B. Vaugon, E. Chailloux, Programming Microcontrollers through High-Level Abstractions, in: Proceedings of the 12th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL 2020, ACM, 2020, p. 5–14. `doi:10.1145/3427765.3428495`.

[14] S. Varoumas, B. Vaugon, E. Chailloux, A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project, in: 9th European Congress on Embedded Real Time Software and Systems (ERTS'18), 2018, pp. 1–10.

[15] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019) 1–84`doi:10.1109/IEEESTD.2019.8766229`.

[16] P. R. Wilson, Uniprocessor Garbage Collection Techniques, in: Y. Bekkers, J. Cohen (Eds.), International Workshop on Memory Management, no. 637 in LNCS, ACM SIGPLAN, Springer, 1992, pp. 1–42. `doi:10.1007/BFb0017182`.

[17] J. Garrigue, Programming with polymorphic variants, in: ML Workshop, 1998, pp. 1–9.

[18] F. Leens, An introduction to I2C and SPI protocols, IEEE Instrumentation Measurement Magazine 12 (1) (2009) 8–13. `doi:10.1109/MIM.2009.4762946`.

[19] S. Varoumas, B. Vaugon, E. Chailloux, Concurrent Programming of Microcontrollers, a Virtual Machine Approach, in: 8th European Congress on Embedded Real Time Software and Systems (ERTS'16), 2016, pp. 1–10.

[20] J. Colaço, M. Pouzet, Clocks as First Class Abstract Types, in: 3rd International Conference on Embedded Software (EMSOFT 2003), Vol. 2855 of Lecture Notes in Computer Science, Springer, 2003, pp. 134–155. `doi:10.1007/978-3-540-45212-6_10`.

[21] S. Varoumas, T. Crolard, WCET of OCaml Bytecode on Microcontrollers: An Automated Method and Its Formalisation, in: Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 5:1–5:12. `doi:10.4230/OASIcs.WCET.2019.5`.

[22] R. Suchocki, S. Kalvala, Microscheme: Functional programming for the Arduino, in: J. Hemann, J. Clements (Eds.), Proceedings of the 2014 Scheme and functional programming workshop, California Polytechnic State University, 2014, pp. 21–29.

[23] R. Gurdeep Singh, C. Scholliers, WARDuino : a dynamic WebAssembly virtual machine for programming microcontrollers, in: MPLR 2019 : proceedings of the 16th ACM SIGPLAN international conference on managed programming languages and runtimes, ACM, 2019, pp. 27–36. `doi:10.1145/3357390.3361029`.

[24] J. Andersen, Programming Arduinos in Ada, Free and Open Source Software Developers' European Meeting (FOSDEM'12) (2012).

[25] M. Grebe, A. Gill, Haskino: A Remote Monad for Programming the Arduino, in: 18th International Symposium on Practical Aspects of Declarative Languages (PADL 2016), Vol. 9585 of Lecture Notes in Computer Science, Springer, 2016, pp. 153–168. `doi:10.1007/978-3-319-28228-2_10`.

[26] B. Shivkumar, J. C. Murphy, L. Ziarek, RTMLton: An SML Runtime for Real-Time Systems, in: Proceedings of the 22nd International Symposium on Practical Aspects of Declarative Languages, Springer, 2020, p. 113–130. `doi:10.1007/978-3-030-39197-3_8`.

[27] A. Sarkar, R. Krook, B. J. Svensson, M. Sheeran, Higher-order concurrency for microcontrollers, in: Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, 2021, pp. 26–35. `doi:10.1145/3475738.3480716`.

[28] J. H. Reppy, Concurrent programming in ML, Cambridge University Press, 2007.

[29] R. Monat, A. Ouadjaout, A. Miné, A multilanguage static analysis of python programs with native C extensions, in: Proceedings of the 28th International Symposium on Static Analysis (SAS 2021), Chicago, IL, USA, Vol. 12913 of Lecture Notes in Computer Science, Springer, 2021, pp. 323–345. `doi:10.1007/978-3-030-88806-0\_16`.

[30] L. Sylvestre, E. Chailloux, J. Sérot, Accelerating OCaml Programs on FPGA, Int. J. Parallel Program. 51 (2–3) (2023) 186–207. `doi:10.1007/s10766-022-00748-z`.

[31] G. Berry, G. Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, Implementation, Science of Computer Programming 19 (2) (1992) 87–152. `doi:10.1016/0167-6423(92)90005-V`.

[32] L. Mandel, M. Pouzet, ReactiveML: a reactive extension to ML, in: P. Barahona, A. P. Felty (Eds.), Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal, ACM, 2005, pp. 82–93. `doi:10.1145/1069774.1069782`.

[33] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, J. Crowcroft, Unikernels: Library Operating Systems for the Cloud, SIGPLAN Not. 48 (4) (2013) 461–472. `doi:10.1145/2499368.2451167`.