SORBONNE UNIVERSITÉ

Sorbonne Université

EDITE de Paris (ED 130)

High-level programming models for resource-constrained microcontrollers

Steven Varoumas

Doctoral thesis in Computer Science

Table of Contents

Introduction

Over the course of an ordinary day, any reader of this thesis will come into contact with numerous microcontrollers. These small electronic devices (ranging from a few millimetres to a few centimetres) are embedded in many of the objects we use every day and quietly govern numerous aspects of our lives. Whether the reader takes public transport to work or drives a personal car, hundreds of microcontrollers accompany the journey, carrying out automated tasks such as controlling the vehicle's braking system or operating the automatic doors on a metro platform. Their presence is not limited to transportation. At work, the coffee machine, the air conditioning system, and even computer peripherals (monitors, mice, keyboards, . . .) may contain dozens of them, each ensuring the proper functioning of these systems. Household appliances are no exception : both small and large devices, such as washing machines, refrigerators, and microwave ovens, rely on a multitude of microcontrollers. In fact, almost every room in a home is affected : alarm clocks, radiators, thermostats, lighting systems, water heaters, and even certain toys may contain them. Some microcontrollers are even carried on one's personinside the mobile phone stowed in one's pocket[1] — or embedded within certain medical devices. Owing to their ubiquity, microcontrollers contribute daily to the many tasks of modern life.

From the inside, a microcontroller (often abbreviated as μC) is a programmable integrated circuit whose various components correspond to those of a highly simplified, yet complete, computer. A microcontroller contains an arithmetic and logic unit, a set of memories – typically consisting of random-access memory for processing the dynamic data of a program, and non-volatile memory used to store the program or certain dataas well as several input/output devices known as *pins* : small metallic legs capable of carrying an electric current to communicate with the surrounding environment (shown in Figure 1). The environment of a microcontroller usually consists of an electronic assembly made up of multiple components that allow interaction with the real world : sensors, which transmit information about the physical environment to the microcontroller (such as temperature values, brightness levels, or the state of a push button, . . .) ; and actuators, which modify the state of the physical system when activated (lighting an LED, rotating a motor, producing sound or heat, . . .). A microcontroller often serves as the central processing unit of such an assembly (or of a subcomponent when dealing with more complex networks of circuits), coordinating the hardware's reactions to the state of its physical environment. For this reason, microcontrollers are frequently used to program embedded systems, governing the automation of tasks specific to those systemsfor example, regulating the temperature of a room.

The physical resources of microcontrollers are often limited, comparable to those of a personal computer from the 1980s. The clock frequency of a microcontroller rarely exceeds a few tens of megahertz, its random-access memory is limited to a few kilobytes, and its flash memory, used to store the program, reaches at most only a few megabytes. These resources may seem strikingly anachronistic in a

1. Although modern smartphones contain more powerful and more complex systems-on-chip, it is not uncommon for microcontrollers to handle auxiliary tasks such as managing touchscreens or certain sensors.

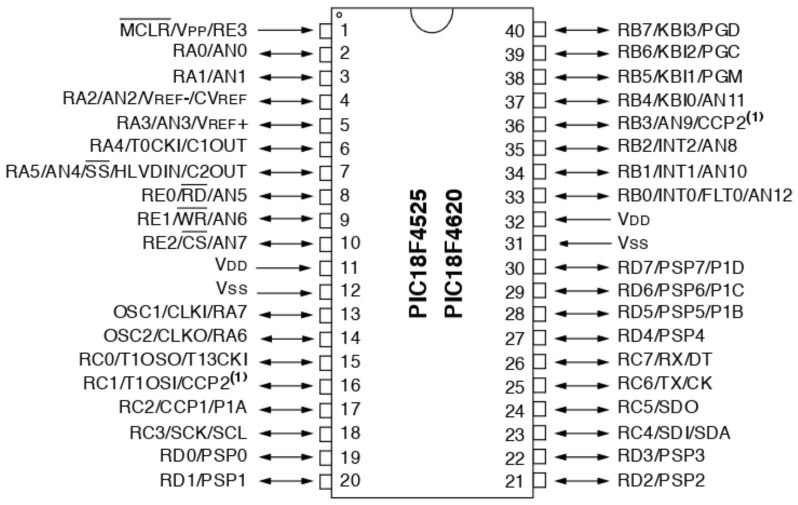


FIGURE 1 – A PIC microcontroller and its input/output pins (numbered 1 to 40)
Double arrows indicate pins that can be used as both input and output. Single arrows indicate pins that can be used only as input (incoming arrow) or only as output (outgoing arrow).

world where even the most modest modern system-on-chip operates a hundred times faster and has one hundred thousand times more memory. Nevertheless, microcontrollers continue to be widely used in industry : their low power consumption and low purchase cost (from a few tenths of a euro to a few tens of euros) contribute to their ubiquity in many everyday objects. In addition, a growing number of hobbyists and do-it-yourself enthusiasts engage in programming microcontrollers, for instance to develop small circuits for home automation (automatic plant watering, lighting control, . . .) or entertainment (handheld game consoles, drones, LED walls, etc.). These uses, encouraged by the emergence of new categories of connected objects, make microcontrollers a frequent choice for developing embedded solutions, whether professional or amateur. Thus, driven by the Internet of Things (*IoT*) — for which some forecasts estimate the number of new devices produced between 2017 and 2025 at one *trillion*[2] [Spa17] — it has been projected that the global microcontroller market would have an annual growth rate of around 12% between 2016 and 2023 [⚓5][3].

Just as the physical resources of microcontrollers resemble those of somewhat outdated conventional computer hardware, the development practices for microcontrollers have also changed very little over time. Microcontrollers are traditionally programmed in languages that can be considered low, or even very low level. It is not unusual, even today, for programs intended to run on microcontrollers to be written directly in assembly language. As a result, programming microcontrollers is a difficult task, requiring the developer to be familiar with the instruction setoften sparse and unexpressiveof the assembly language specific to the target microcontroller. The programmer must also know the hardware on which the program will run in detail, since the level of hardware abstraction provided by such languages is very limited, if not nonexistent. This extreme specialization of programs for microcontrollers makes them hard to approach for novice programmers accustomed to higher-level languages such as Python or Java, and it restricts portability : even a minor change of microcontroller model in the target application may require rewriting the program code from scratch. Testing and debugging microcontroller programs is equally

2. One trillion = 10^{12}.
3. All web references in this document are preceded by a symbol representing an anchor ⚓.

complex and restrictive. While debugging on a personal computer is facilitated by software debuggers that allow step-by-step simulation of execution while monitoring memory contents, and while running a test suite on a PC can be done with a simple mouse click, development environments for microcontrollers do not always provide the possibility of simulating execution. Moreover, such simulation can be difficult, since the program is closely tied to its physical execution environment. Consequently, testing a program intended for a microcontroller often means executing it directly on the actual electronic assembly for which it was designed. Debugging, especially for hobbyist developers, often involves compiling the program, transferring the generated executable onto the microcontroller, and simply observing whether it behaves as expected within the real hardware assembly. This process may have to be repeated many times until the program *appears* to work correctly. Such a tedious and unreliable practice is not only extremely time-consuming but can also damage the electronic hardware in use -- for example, through wear of the microcontroller's limited non-volatile memory (which supports only a finite number of write operations), or even through destruction of peripheral components (or the microcontroller's own internal components) if the program interacts with them incorrectly.

Within certain communities of developers familiar with microcontroller programming, the mere use of a subset of the C language for embedded systems is sometimes regarded as high-level programming. Compared to assembly languages, C provides a significant layer of hardware abstraction that simplifies development and debugging, while also offering certain guarantees for programs (including simple static type checks) that make them safer and less prone to bugs. As a result, the use of relatively low-level languages such as C often represents a boundary rarely crossed by embedded systems developers, who wish to retain precise control over resource consumption and hardware interactions in their programs. Yet many higher-level languages offer advantages that are particularly valuable in the context of microcontroller programming. Beyond the increased expressiveness provided by paradigms such as object-oriented or functional programming, the guarantees afforded by such languages (for instance, static or dynamic type checking) constitute a clear benefit when programming embedded systems whose malfunctions can have disastrous consequences. Moreover, the hardware abstractions provided by these languages do not necessarily conflict with the careful use of limited resources. Since microcontrollers offer little memory, it can actually be advantageous in certain cases to rely on a runtime environment that dynamically reuses resources according to the program's needs. For these reasons, several efforts have been made to enable the development of embedded programs in higher-level programming languages such as Java, Python, or Scheme. Other approaches propose specialized languages for microcontroller programming inspired by higher-level paradigms, such as reactive functional programming [HG16] or graphical programming [Kat10, MS17].

Moreover, general-purpose languages commonly used in programming are not necessarily suited to the specific characteristics of developing applications for microcontrollers. Since an embedded system must generally react quickly to various stimuli (such as a button press or a change in a sensor reading), microcontroller programming often exposes concurrent behavior. Providing a lightweight concurrency model tailored to such applications is therefore a significant advantage, as it simplifies the development process. However, the concurrency model chosen for programming a microcontroller cannot systematically mirror those commonly used in programming applications for computers or mobile phones (such as thread-based systems) because of the limited resources of microcontrollers and their intrinsic features, notably the absence of an operating system. Less common concurrency models are better suited to microcontroller programming. In particular, the synchronous programming model appears especially appropriate, given its ability to function with very few hardware resources and its simple foundation on the *synchronous hypothesis*, which assumes that all components of a program execute *instantaneously* and concurrently. This paradigm has the advantage of freeing the developer from concerns related to synchronization between the various components of a program, thereby contributing to a higher level of abstraction in the software development process.

Furthermore, programs designed for embedded systems—sometimes safety-critical—often need to meet strict physical or behavioral constraints. Certain microcontroller applications (such as drones, transportation systems, or specific robots) correspond to *real-time systems*, in which reactions to external stimuli must occur within a given time interval. This imposed deadline is generally tied to safety requirements, and ensuring the system's reaction speed is an essential condition for the program to behave as intended. Other constraints may concern the logical behavior of programs, for which certain invariants—representing program correctness—can be verified. We draw partly on development practices from civil avionics, where industrial tools derived from synchronous languages, such as SCADE [CPP17], enable DO-178C certification of aircraft (and their embedded software). The use of formal methods (abstract interpretation, *model checking*, . . .) to verify certain program properties is also encouraged. For example, it is now possible to replace some tests with proofs under the DO-178 standard [4]. Such robust analyses are of clear interest in the programming of critical embedded systems. Our approach, combining high-level programming models with the synchronous dataflow paradigm, incorporates similar analyses to verify program correctness and to check properties such as the computation time of a synchronous instant.

The ambition of this thesis is to propose a solution that bridges two worlds which may appear far apart. On one side, the world of microcontroller programming, rich in relatively simple applications and widely used by both hobbyists and industry, but constrained by limited hardware resources and development practices that often provide little in the way of safety or software guarantees. On the other side, the world of higher-level programming models, which offer abstractions that allow greater expressiveness and stronger safety, but tend to produce programs that are potentially more resource-intensive. The central ambition of our approach lies in providing microcontroller developers with modern, powerful development techniques that enhance the guarantees associated with their programs while still accounting for the limited capacity of the hardware in use. Our solutions are therefore designed to be executable on extremely resource-constrained devices, with only a few kilobytes of RAM. This focus on compatibility with limited-resource microcontrollers reflects the reality that, even today, 8-bit microcontrollers equipped with just a few kilobytes of RAM continue to dominate the market [SSD+17]. Moreover, ensuring that our solutions function on such constrained hardware guarantees that our approach is applicable to a wide range of microcontroller models, not just the more powerful—but less commonly used—higher-end variants.

This manuscript details our solution, which is based on a series of abstractions aimed at simplifying development processes and ensuring the correctness of programs for embedded systems. In particular, we will focus on formalizing several aspects of our approach and on proving certain metatheoretical properties of our solution. Much of this formalization has been mechanized through software tools, and several proofs relating to this formalism were carried out using the Coq proof assistant [Tea19]. The source

4. Additional information on this subject is available in a document titled DO-333

code of the programs, examples, and the proofs of the lemmas and theorems presented throughout this manuscript are available online [⚓1].

Structure of the manuscript

— Chapter 1 presents the classical methods of microcontroller programming, as well as prior work aimed at increasing the expressiveness and safety of microcontroller programming. We review the state of the art of the synchronous programming model, as well as classical techniques for providing guarantees on synchronous programs in the field of safety-critical embedded systems.

— Chapter 2 introduces our implementation of the OCaml virtual machine, called OMicroB. This virtual machine is designed to run on microcontrollers with severe memory constraints. Through its portability, and by leveraging the specific features of the OCaml language and its runtime library, OMicroB provides a first level of abstraction that enables more expressive and safer programming of microcontrollers.

— In Chapter 3, we propose OCaLustre, a synchronous dataflow extension of the OCaml language. Designed for microcontroller programming and inspired by the Lustre language, this extension offers a lightweight programming model for developing concurrent behaviors in embedded systems. After presenting an overview of the language features, this chapter describes the formal specification aspects of OCaLustre.

— In Chapter 4, we describe the main compilation steps of OCaLustre programs. This compilation process transforms synchronous code into a sequential OCaml program, fully compatible with any OCaml compiler, while verifying several static guarantees related to typing and the scheduling of software components.

— In Chapter 5, we present methods for the formalization and verification of various properties derived from the language specification. We describe in particular methods that verify the consistency of an OCaLustre program with two type systems, whose rules define the correct semantics of a program.

— Chapter 6 describes a method for computing the worst-case execution time (WCET) of an OCaLustre program. This method leverages the portability of our approach by analyzing the *bytecode* file generated after compilation. We present the proof of correctness of this process before describing the software prototype that implements it.

— Chapter 7 provides a performance analysis of the different software solutions presented in this manuscript. Based on several example programs, each implementing various features of the OCaml language, we evaluate the memory footprint and execution speed of the OMicroB virtual machine. We then highlight the low memory footprint of the OCaLustre synchronous extension and analyze its execution performance.

— Chapter 8 presents several concrete applications that showcase the advantages of our different abstraction levels and of verifying guarantees on developed programs. These examples particularly demonstrate the ease of describing the interactions between a microcontroller and its environment, and they confirm the adequacy of our chosen model with the limited resources of the target devices.

— Finally, we conclude this thesis by summarizing the various approaches proposed in our work, and by discussing possible extensions to our solutions aimed at further increasing the level of abstraction and safety in the models considered.

1 Preliminaries

The different layers of abstraction proposed in this thesis are based on technical and technological choices motivated by practical considerations. This chapter provides an overview of various state-of-the-art solutions that correspond to each of the abstraction levels considered. As we review these works, we justify the choices made in this thesis to pursue one direction rather than another. We begin by examining the physical composition of a microcontroller and the classical programming methods used for developing embedded applications. We then explain our motivation to abstract the underlying hardware through a virtual machine for a high-level language—one that is both expressive and has a runtime environment with a small memory footprint. Our desire to provide a concurrency model that is simple and lightweight is supported by our choice of a synchronous dataflow programming model, well suited to the nature of microcontroller programs. Finally, we discuss different ways of verifying and guaranteeing certain properties of the programs developed using these abstractions.

1.1 Physical and software characteristics of microcontrollers

To give the reader a sense of the specific challenges and techniques involved in microcontroller programming, this section presents a general overview of the main hardware aspects of such generic integrated circuits. We first describe the structure of a generic microcontroller, before addressing its technical characteristics, and in particular its memory limitations. We also discuss classical programming methods for microcontrollers, the development environments associated with them, and the limitations of these development processes.

1.1.1 Composition and resources of a microcontroller

A microcontroller is composed of several elements that enable it to perform the computations required for running its dedicated programs and for interacting with the electronic circuit that forms its environment [BV97]. The main components of a microcontroller are as follows :

— A central processing unit, or *CPU* (Central Processing Unit), whose role is to perform the arithmetic and logical operations required for executing a program. The processing speed of a resource-constrained microcontroller is generally measured in tens of MIPS[1].

— Random-access memory, or *RAM*, which contains the volatile, dynamic data generated during program execution. In the context of this thesis, RAM is certainly the most limited resource : typically only a few kilobytes on the microcontrollers we consider, it forces the developer to carefully manage the memory consumption of their program.

— Non-volatile flash memory, used to store the program code. Although flash memory can technically be written during program execution, it is usually treated as read-only memory (or *ROM*)—both to separate code from data and because of its physical limitations. Flash memory supports only

1. Millions of instructions per second.

a limited number of write cycles. For instance, the flash memory of an ATmega microcontroller can withstand (according to the manufacturer's specifications) about 10,000 write cycles, whereas its RAM is virtually indestructible. The size of flash memory is generally ten to a hundred times larger than that of RAM. On the microcontrollers we consider, it typically ranges from a few dozen to a few hundred kilobytes ; for example, the ATmega328P has 32 KB of flash memory.

— Timers, which allow durations to be measured and synchronize the various electronic components of a circuit with one another. These timers are incremented at regular time intervals. An internal clock signal, triggered at each program instruction, determines the rate at which the timers are incremented.

— An interrupt mechanism that, upon the occurrence of a specific internal or external stimulus, immediately triggers the execution of a dedicated routine. External interrupts can be caused by a change in an electrical signal on a pin (for example, pressing a push button), while internal interrupts may be triggered by the overflow of a timer or by pulses from an internal electronic oscillator.

— Input/output ports that allow communication with electronic components connected to the microcontroller. This communication occurs by exchanging electrical signals of varying voltage (e.g., 0 or 5 Volts), representing the transmission of binary signals between the microcontroller and its environment. These ports correspond to a set of metallic pins visible from the outside of the microcontroller, onto which wires or printed circuits are soldered to connect them to external components. Each pin of a port must usually be configured (typically by setting a bit in the register corresponding to the relevant port) at the beginning of the program to declare it as either an input or an output interface.

— Some input/output ports also support the reading of analog values through *analog-to-digital converters* (*ADC*). These can translate time-sampled voltage variations on a pin into an analog value. Similarly, the translation of an internal digital signal into an external analog signal is made possible by the presence of a *digital-to-analog converter* (*DAC*).

Figure 1.1 provides a simplified schematic representation of the relationships between the main elements of a microcontroller. The interactions between these elements are primarily carried out via transfers on one (or more) bidirectional buses, which, for instance, convey program instructions from flash memory to the CPU, or write data resulting from a computation into RAM.

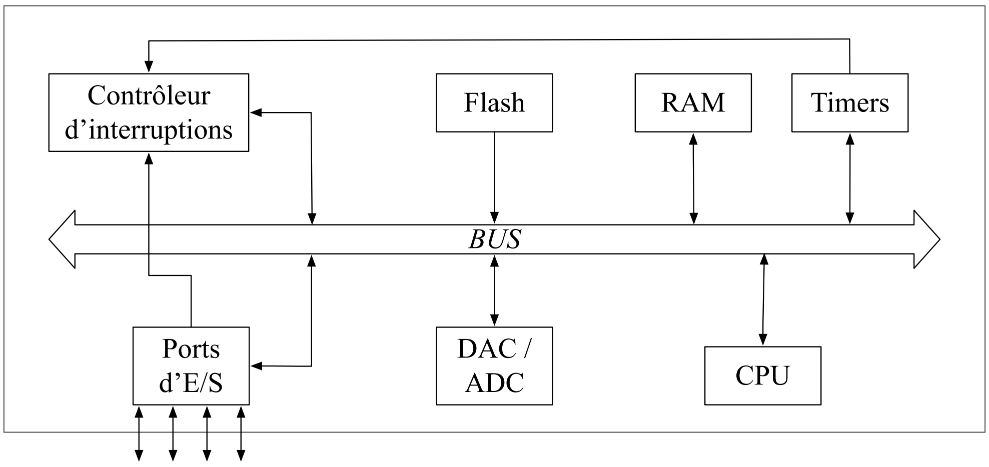


FIGURE 1.1 – Internal structure (simplified) of a microcontroller

Several families of microcontrollers are currently available on the market. Among them, AVR microcontrollers are widely used by hobbyist programmers thanks to their presence in Arduino/Genuino boards [⚓15]. These boards, which integrate a microcontroller along with pre-connected electronic components (a USB port, a reset button, a power connector, etc.), simplify the process of developing embedded programs. The Arduino Uno board [Bad14], for instance, contains an AVR ATmega328 microcontroller, equipped with 2 kilobytes of RAM and 32 kilobytes of flash memory. Other families of microcontrollers with different architectures are also available, such as the STM32 line based on an ARM architecture, which often comes with more substantial resources. PIC microcontrollers, though less well known among hobbyists, are very commonly used in industry due to their low cost and efficiency.

As an example, the table in Figure 1.2 summarizes the physical characteristics of a selection of microcontrollers from several families—some with extremely limited resources (less than 1 kilobyte of RAM), and others approaching the capabilities of personal computers from the 1990s. In our context, we focus on microcontrollers closer to the lower end of this spectrum, with relatively little flash memory (less than 100 kilobytes) and very limited RAM (less than 8 kilobytes), such as the PIC 18F4620.

Model	Architecture	Flash memory (KB)	RAM (B)	CPU speed (MIPS)	Operating voltage
AT89C51	Intel 8051 – 8 bit	4	128	12	4 to 6 V
ATmega328P	AVR – 8 bit	32	2048	20	1.8 to 5.5 V
ATmega2560	AVR – 8 bit	256	8192	16	1.8 to 5.5 V
PIC 18F4620	PIC – 8 bit	64	4096	10	2 to 5.5 V
PIC 24FJ128GA006	PIC – 16 bit	128	8192	16	2 to 3.6 V
STM32 L051C8	ARM – 32 bit	64	8192	32	1.65 to 3.6 V
STM32 F091VCT6	ARM – 32 bit	256	32768	48	2 to 3.6 V

FIGURE 1.2 – Some microcontrollers and their characteristics

A microcontroller is therefore structurally very close to a simplified representation of a personal computer, but with far fewer resources. Its applications, however, differ greatly from those of conventional PCs or even single-board computers (such as the *Raspberry Pi*) : it generally has neither an operating system, nor a file system, nor standard peripherals such as a keyboard or a mouse. A microcontroller is typically dedicated to controlling an automated task, most often in an *embedded* context where it is connected only to an electronic circuit specific to that task, and where an application is executed « bare-metal »—with no software intermediary between the hardware and the program.

1.1.2 Classical programming models for microcontrollers

Low-level programming languages

Developing an application for a microcontroller is a fairly complex task, requiring the developer to have a solid understanding of the hardware in use. In an embedded system, the close relationship between programs and hardware traditionally leads to the use of low-level programming languages, which offer little or no abstraction from the hardware for application development. Applications for embedded systems are often written in assembly language, in order to finely control hardware configuration and resource consumption. As a result, the choice of microcontroller has a major influence on the development process, since different families of available microcontrollers do not share the same assembly instruction set. For example, a program written for a PIC microcontroller cannot be executed on an AVR microcontroller, and vice versa. Even physical differences between microcontrollers of the same family can limit the portability of a program.

Relatively more portable, subsets of the C language are also widely used for programming embedded applications. The greater expressiveness of C compared to assembly is valued by embedded systems developers, while its fine-grained control over memory resources is considered essential for applications intended to run on resource-constrained devices. Nevertheless, programs traditionally written for microcontrollers remain limited in terms of hardware abstraction, portability, and static verification of program properties—such as type correctness.

For example, Figure 1.3 shows the source code of a C program for an AVR ATmega328P microcontroller. This program emits an electrical pulse at regular intervals on pin PB5 (i.e., pin number 5 of port B of the microcontroller), in order to make a light-emitting diode (*LED*) connected to it blink[2].

```c
#ifndef F_CPU
#define F_CPU 20000000UL // clock speed 20 MHz
#endif

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  DDRB = DDRB | 0b00001000; // Configure PB5 as output
  while(1)
  {
    PORTB = PORTB | 0b00001000; // Turn PB5 on
    _delay_ms(500); // wait half a second
    PORTB = PORTB ^ 0b00001000; // Turn PB5 off
    _delay_ms(500); // wait half a second
  }
}
```

FIGURE 1.3 – A C program for the ATmega328P microcontroller

It is important to note here the relatively poor readability of such a program : interactions with input/output ports are carried out through binary operations that, using masks, modify the bits of registers representing the program's ports (`DDRB` configures the pins of port B as inputs or outputs, and `PORTB` modifies their values). In a more complex program, even a small error in such operations can be difficult for the developer to detect and may lead to malfunctions that could damage the electronic circuit.

Some software libraries, such as the Arduino library, provide C primitives that slightly abstract these operations (for example, through the use of a function like `digitalWrite()`, which takes as parameters the name of a pin and the signal to emit). However, these remain quite limited and are essentially comparable to simple macros, making programs somewhat easier to read and write, but without providing any additional safety guarantees for the resulting code.

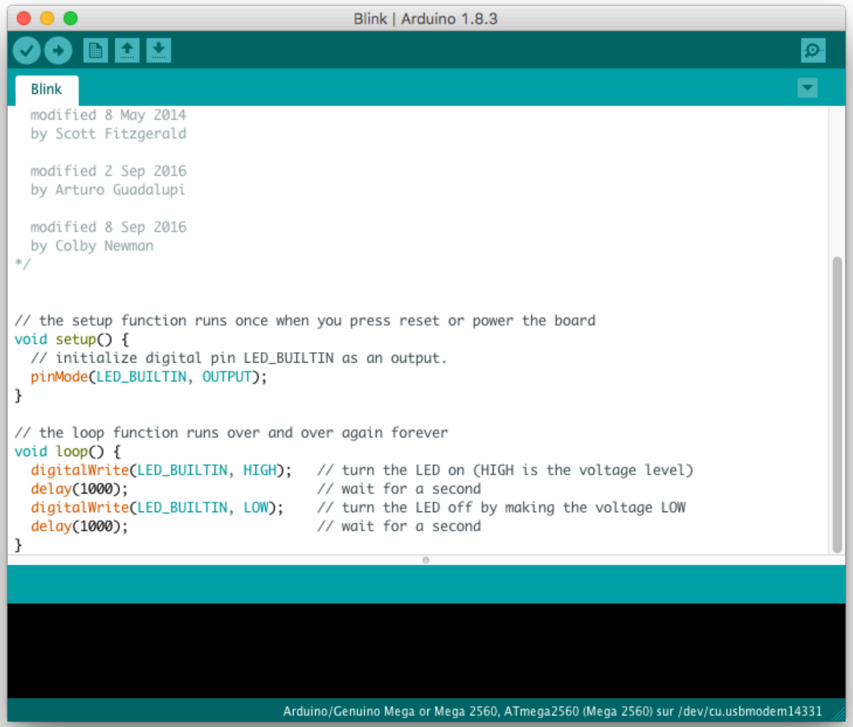
The portability of such a program is also very limited. For example, some AVR microcontrollers do not use the same port names, and porting this program to a microcontroller from another family requires

2. In C, the operator ˆ corresponds to the bitwise exclusive or between two bits.

significant changes specific to the chosen microcontroller. As an illustration, port register configurations differ between AVR and PIC microcontrollers : a bit set to 1 in a port configuration register designates an output pin on an AVR microcontroller, whereas it designates an input pin on a PIC.

Development environments

The development environments used to create and debug programs for microcontrollers are themselves complex and relatively inflexible. A few proprietary, generally single-platform solutions are provided by manufacturers. For example, the integrated development environment (*IDE*) MPLAB [⚓10] allows programming, simulation, and transfer of programs to PIC microcontrollers. *AtmelStudio* [⚓8], meanwhile, is an IDE for AVR microcontrollers that offers more or less the same features as MPLAB. Finally, the *Arduino IDE*[⚓7] is a cross-platform application that allows developers to write programs for Arduino development boards, compile them, and transfer the generated executables onto the board. Figure 1.4 shows the very simple user interface of this software.



```
  modified 8 May 2014
  by Scott Fitzgerald

  modified 2 Sep 2016
  by Arturo Guadalupi

  modified 8 Sep 2016
  by Colby Newman
*/


// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

Arduino/Genuino Mega or Mega 2560, ATmega2560 (Mega 2560) sur /dev/cu.usbmodem14331

FIGURE 1.4 – The Arduino integrated development environment (IDE)

These mainstream tools are for the most part fairly modest in functionality : they can generate an executable for the microcontroller but do not necessarily allow proper testing of the program, since the behavior of such a program is intrinsically linked to the electronic circuit that surrounds it. More powerful software solutions, such as the Proteus suite [⚓17], allow computer-assisted composition of electronic circuits and simulation of microcontroller programs in executable form. Unfortunately, such software is very expensive and relatively difficult to use, making it largely inaccessible to non-specialist developers. It is therefore not uncommon for programmers of small embedded applications to rely instead on an *in-situ* debugging method, using the physical microcontroller directly to test their program : with a serial connection to a computer, they analyze the program's responses to various stimuli. This method of debugging is long and tedious, and several errors in the program code—which could have been detected prior to transfer to the physical hardware—may slow down the development cycle.

Programmers wishing to turn to *open source* solutions can use a handful of free C compilers, most of them based on GCC, which are available on multiple platforms. For example, the avr-gcc compiler [⚓25], together with the avr-libc standard library and the avr-binutils toolchain, provides a complete compilation environment for AVR microcontrollers. The generated executable program can then be transferred with the avrdude tool (*AVR Downloader UploaDEr*), which is distributed with the compiler. On the PIC side, ongoing work in the *sdcc* compiler [⚓27] enables the translation of C source code for devices in the PIC16 and PIC18 families. The *usbpicprog* tool [⚓23], meanwhile, is an *open source* chip programmer (i.e., a device that transfers a program from a computer into the flash memory of a microcontroller) for these microcontrollers.

Microcontrollers are programmable integrated circuits equipped with limited resources and development environments. Programming methods for microcontrollers generally reflect these constraints and are therefore traditionally based on the use of low-level languages. As a result, such languages can discourage inexperienced developers who are unfamiliar with these technologies. Developing in assembly or C often leads to programs that are relatively long to write and that may also contain errors that are difficult to detect.

It therefore seems appropriate to explore different programming models, inspired by the abstractions used in developing applications for personal computers, with the goal of providing microcontroller programming with tools that are easier to grasp. In particular, we are interested in the use of so-called *high-level* languages for programming embedded applications.

1.2 Hardware abstraction and high-level languages

It is now very rare, in the more common context of programming for personal computers (or for modern equivalent devices such as smartphones or tablets), for programs to be written directly in assembly languages. Similarly, the use of the C language is increasingly limited to specific domains, such as the development of system applications that, by their nature, must manipulate memory directly. Today, it is commonplace to write applications with which users interact directly (mobile applications, web applications, . . .) in high-level languages, which facilitate the development of complex applications.

High-level programming languages are those whose expressive power abstracts away from the computational model of the machine on which programs are executed. They provide rich control structures and implement diverse programming paradigms that can manipulate, for example, functional values, objects, exceptions, or continuations. These languages natively offer rich data structures, making it easier to build complex applications. Several high-level programming languages rely on static typing to improve program safety and are based on runtime environments that provide automatic memory management. The debugging process of applications written in these languages is simplified by the use of symbolic debuggers, which handle the complex data structures of the language. Finally, these high-level languages are equipped with mechanisms that enable interoperability with low-level languages, thus allowing direct interfacing with hardware.

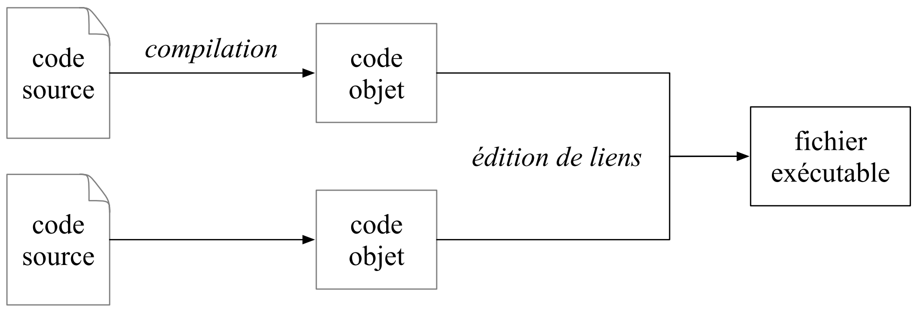


Figure 1.5 – Compilation into a native program

It therefore seems natural to consider using such languages for programming embedded systems in order to bring the same advantages to microcontroller development, in place of the traditional assembly/C pair. In this section, we also address the compilation of these high-level languages into non-native code, interpretable by a *virtual machine* (or *abstract machine*) positioned between the program and the hardware. In the following, we describe various experiments and state-of-the-art systems that enable high-level programming on microcontrollers through this *virtual machine approach*.

1.2.1 The virtual machine approach

In a highly simplified view, the classical compilation process of a program consists of translating its source code into native code, capable of being understood by the hardware on which it will be executed. Figure 1.5 illustrates such a mechanism : source files are translated into the native language of the target machine. In the case of low-level programming languages, this translation is relatively straightforward, given the proximity between the language instructions and those of the hardware. Assembly is simply a human-readable version of the processor's binary instructions, and compiling a C program is fairly direct (at least when one does not attempt to optimize the generated code), since the imperative control operators of the language closely resemble those of the hardware.

Compiling programs that implement high-level programming language features (functional programming, object-oriented programming, . . .) is a more complex process. Since physical processors are only capable of natively handling simple imperative operations, multiple transformations must be applied during compilation in order to convert high-level features into purely imperative native code. Compilers for high-level languages therefore manipulate numerous intermediate representations of programs, and these representations remain the same whether the final program is intended to run on hardware with architecture A or on hardware with a completely different architecture B.

The virtual machine approach consists of carrying the compilation process only up to the production of a common, non-native representation of programs : the *bytecode*. It is then the responsibility of a *virtual machine*—that is, a bytecode *interpreter* associated with a *runtime environment*—to execute the program in this intermediate form [DS00]. This approach enables the *portability* of applications, since any program translated into the language's bytecode can be executed on any device that provides an interpreter for this bytecode. The use of virtual machines for executing rich programs was popularized by the Java language and its JVM (*Java Virtual Machine*), whose motto *Write once, run anywhere*[3] highlighted the ability of the same Java bytecode to run on a wide variety of platforms.

3. *Écrire une fois, exécuter partout*

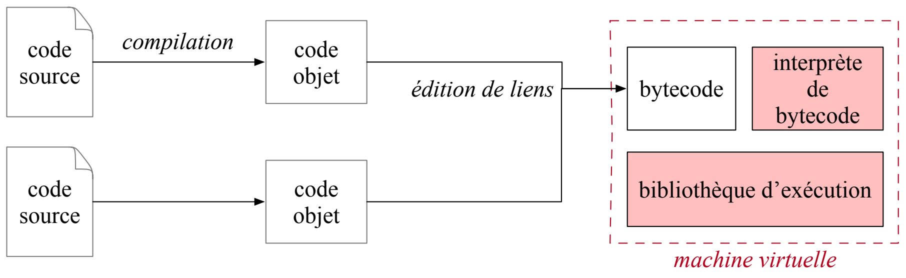


FIGURE 1.6 – Virtual machine approach : compilation into *bytecode* and interpretation

Figure 1.6 illustrates how the source code of a program is compiled to be interpreted by a virtual machine.

The *bytecode* of a high-level language is composed of instructions that are richer than native machine code. As a result, a single *bytecode* instruction typically corresponds to a sequence of machine-language instructions. A program translated into bytecode can therefore, thanks to this *factorization*, be more compact than its equivalent in machine code. Of course, bytecode cannot be executed without its runtime environment, and the size of this environment must also be taken into account in order to make a fair comparison. Nevertheless, since the runtime size is fixed, the total memory footprint of a bytecode program plus its virtual machine can, in the case of substantial programs, be smaller than that of a program compiled into native machine code. Such compaction of the total program size is an obvious advantage in our context, where memory resources must be used sparingly.

Moreover, using a common bytecode across all execution platforms also allows the factorization of several program analyses, such as memory resource estimation [AGG07], computation time estimation [SP06], vulnerability detection [LL05], or even plagiarism detection [JWC08]. Even if some analyses may lose precision due to this factorization [LF08], the benefit remains considerable in our use case, given the wide variety of microcontroller architectures and models.

1.2.2 Programming microcontrollers in high-level languages

A large number of projects aiming to execute high-level languages on microcontrollers have been developed in recent years. In this section, we examine a sample of these projects, which enable the development of programs that implement diverse programming paradigms such as object-oriented programming or functional programming. Some of these solutions rely on native compilation models, which generate machine code directly from a program's source code, while others benefit from the advantages of the virtual machine approach and its common bytecode representation.

High-level languages and native compilation models for microcontrollers

Using a high-level language does not necessarily require a virtual machine and a bytecode interpreter. Several solutions that enable programs written in high-level languages to run on microcontrollers are based on a compilation model that produces native code, which can be executed directly on the target hardware without any software intermediary. Among these, it is possible to write programs in C++, a multi-paradigm language whose proximity to C allows for good performance, for example through the *avr-g++* compiler for AVR microcontrollers, or the *MPLAB XC32++* compiler, which supports C++ compilation but only for 32-bit PIC microcontrollers. The *IAR Embedded Workbench* compiler, meanwhile, allows C++ programs to be compiled for a variety of microcontroller families, including MSP microcontrollers, AVR microcontrollers, and STM32 microcontrollers. Nevertheless, C++ does not provide all the abstractions offered by higher-level languages, such as automatic memory management or a safer type system.

Several projects have also explored running programs written in the Ada language on microcontrollers. Ada is an object-oriented programming language widely used in safety-critical embedded systems (automotive, aerospace, etc.) [Reg12]. However, some of these efforts cannot offer the full benefits of the language because of their significant memory footprint [RP19]. For instance, the AVR-Ada compiler project [⚓24] is compatible with resource-constrained 8-bit AVR microcontrollers, but it does not support, for example, the *Ravenscar* profile designed for programming safe real-time systems. The *GNAT GPL* compiler developed by Adacore, on the other hand, supports all language profiles but targets only ARM-based microcontrollers, which have far greater resources.

Other high-level languages, such as Rust, are compiled into a common intermediate representation before being translated into the machine-specific language. This representation, the *bitcode* of the LLVM compiler infrastructure (historically standing for *Low-Level Virtual Machine*), is equivalent to a generic low-level bytecode that can be translated into machine code for each target [Lop09]. This *bitcode* could also be interpreted, although this approach is fairly uncommon. For example, a university project that implemented a *bitcode* interpreter for MSP430 microcontrollers represented an original attempt [Cam16], though the prototype suffered from noticeable slowness. Other ongoing work includes compiling Rust programs to AVR microcontrollers using the LLVM backend for AVR [⚓26]. Another project aims to execute programs written in a subset of the *Go* language on any target supported by LLVM [⚓28]. Unfortunately, microcontroller support within the LLVM project remains very limited : apart from the ongoing work on AVR, only certain ARM-based microcontrollers are supported, and support for PIC microcontrollers, for instance, was abandoned in 2011.

High-level languages and their virtual machines on microcontrollers

Solutions based on a virtual machine approach appear to us to be the most effective way of providing, in a straightforward manner, a higher-level programming model for resource-constrained microcontrollers. Such approaches increase the portability of programs, thanks to the genericity of the bytecode generated during compilation, which abstracts away from the specific hardware on which the virtual machine is executed. To illustrate the advantages of this approach, in this section we describe a representative sample of different projects that implement virtual machines for microcontrollers from various families.

Java on AVR and MSP : Thanks to its popularity, the richness of its applications, and its philosophy of program portability, the Java language is an obvious candidate for the use of a high-level language on microcontrollers. As a result, several projects and experiments have been conducted to port the Java Virtual Machine (*JVM* [LYBB14]). These projects enable embedded application developers to benefit from Java's rich language features, which include object-oriented, imperative, and more recently functional programming paradigms, along with its static typing with nominal subtyping, and the extensive class library provided by the language (the Java *API*), which makes it easy to represent advanced data structures.

For example, the industrial solution microEJ [⚓16] allows Java programs to run on development boards containing microcontrollers, most of them based on ARM architectures. The Darjeeling system [BCL08], meanwhile, provides a multi-threaded Java virtual machine capable of executing Java bytecode on AVR128 and MSP430 microcontrollers, which contain between 1 and 8 kilobytes of RAM and between 16 and 128 kilobytes of flash memory. Other solutions, such as HaikuVM [⚓13] or NanoVM [⚓14], enable the execution of Java bytecode on microcontrollers used in Arduino development boards.

However, the richness of the Java language can sometimes be incompatible with these solutions designed for resource-constrained machines. Java manipulates relatively large data structures, themselves composed of many objects. A program written in a standard Java style, making heavy use of objects and complex data types, can consume large amounts of memory that may simply not be available on such devices. The example of *Java Card* [Gut97], a system designed to execute Java programs on smart cards, illustrates this difficulty : version 2 of Java Card can run on hardware with only 2 KB of RAM and 64 KB of ROM, but lacks a garbage collector, does not support multitasking, and is restricted to relatively simple data types (for instance, it does not support multidimensional arrays or collection classes). Version 3 of Java Card addresses these shortcomings, but at the cost of much higher resource consumption : it requires 24 KB of RAM and at least 128 KB of ROM. Similarly, the TinyVM project [HPK$^+$09], initially designed to create a low-footprint Java virtual machine for RCX microcontrollers (used in programmable *Lego Mindstorm* bricks) for sensor network development, was eventually integrated into the LeJOS project [LHS10]. While LeJOS offered richer functionality, it was also more resource-hungry.

Python on STM32 : MicroPython [Bel17] is an implementation of the Python 3 language designed primarily to run on the *pyboard*, which contains an STM32F405RG microcontroller with a Cortex-M4 processor clocked at 168 MHz, 1024 KB of flash memory, and 192 KB of RAM. MicroPython allows the execution of Python 3 programs, offering the advantages and features of the language : its clear syntax, accessible even to beginners, its object-oriented design, and certain high-level features (such as list comprehensions) that enable the concise development of complex programs. MicroPython is based on *CPython*, the reference implementation of the language, which provides a bytecode interpreter for translated Python programs. A particularly interesting feature of MicroPython is its ability to communicate with a *REPL* (*Read-Eval-Print-Loop*) running on the microcontroller itself [VVF18]. This feature enables quick testing and facilitates debugging.

MicroPython targets microcontrollers with relatively rich resources, well beyond the devices considered in this thesis. The lower limit of technical capabilities required for running the MicroPython interpreter is about 16 KB of RAM and 256 KB of program memory. Other work has also produced Python interpreters for microcontrollers [BSR12], but these too target more capable devices : Cortex-M3 microcontrollers clocked at least at 50 MHz, with at least 64 KB of RAM and 512 KB of flash memory. Finally, the projects *python-on-a-chip* [⚓12], followed by *PyMite* [Hal03], now discontinued, allowed Python programs to run on smaller microcontrollers (with about 50 KB of flash memory and less than 8 KB of RAM recommended for program execution). However, this came at the cost of severe limitations : only a subset of Python 2.5 was supported, and no standard library was provided. Overall, the body of work on Python also illustrates the difficulty of porting all the features of a multi-paradigm language to hardware with very limited resources.

Scheme on PIC : Scheme is a functional programming language derived from Lisp. Scheme is highly expressive, thanks for example to its *macro* system, as well as its explicit manipulation of continuations in programs via the *call-with-current-continuation* (*call/cc*) construct. There exist numerous implementations of Scheme targeting personal computers. These implementations are based on specifications (the so-called *Revised Reports on the Algorithmic Language Scheme* or *RnRS*, where *n* corresponds to the revision number) that define the features of the language. Not all of these implementations (for instance *Bigloo* [⚓18]) rely on a virtual machine, but some, such as *Guile* [⚓22] (an implementation of Scheme conforming to the R6RS standard), do so by translating Scheme code into bytecode composed of 175 different instructions, executed by an interpreter written in C.

A few virtual machines capable of executing subsets of the Scheme language on resource-constrained microcontrollers have been developed. Among them, BIT [DF05] and PICBIT [FD03] enable execution of the R4RS standard on microcontrollers with less than 8 KB of RAM and 64 KB of program memory, while the PICOBIT system [SF09] supports Scheme programs written in subsets of R5RS, executable on PIC18 devices with as little as 1 KB of RAM and 6 KB of ROM. Because of its lightweight nature, Scheme thus appears particularly well-suited to resource-limited devices.

OCaml on PIC : OCaml is a language that combines multiple programming paradigms. Since its inception, it has incorporated features of object-oriented, functional, modular, and imperative programming. This diversity of paradigms allows expressive and convenient development of complex programs. Its strong static type system, combined with type inference, ensures at compile time that programs are free of inconsistencies in the use of typed values.

The standard virtual machine of OCaml, known as the *ZAM* and derived from the *ZINC* machine [Ler90], is a stack-based machine built on a uniform representation of data. The bytecode associated with the OCaml VM consists of 148 instructions, which include classical operators for computation and branching, as well as instructions dedicated to the manipulation and application of functional values.

The OCaPIC project [VWC15] is an implementation of the standard OCaml virtual machine for programming PIC18 microcontrollers. OCaPIC makes it possible to execute the full OCaml language on microcontrollers with very limited resources (4 KB of RAM and 64 KB of flash memory). This ability to run the entire OCaml language on such constrained devices is remarkable, and it highlights both the lightweight nature of the OCaml virtual machine and the strength of the optimizations provided by this implementation.

In addition, OCaPIC provides several tools to improve the development of OCaml programs for microcontrollers, including in particular *ocamlclean*, a tool that statically removes unused closure allocations from programs, as well as a set of simulators that simplify debugging of developed programs.

Figure 1.7 is a Venn diagram representing the various technologies discussed in this section.

The table in Figure 1.8 summarizes the main characteristics of the approaches presented. Virtual machine–based approaches have several advantages for microcontroller programming. They provide features from high-level languages, with which modern application developers are familiar, while also potentially reducing the memory footprint of programs thanks to the factorization inherent in representing programs as bytecode. Nevertheless, some of these virtual machines are not particularly well suited to use on resource-constrained microcontrollers, which may have only a few kilobytes of RAM and less than a

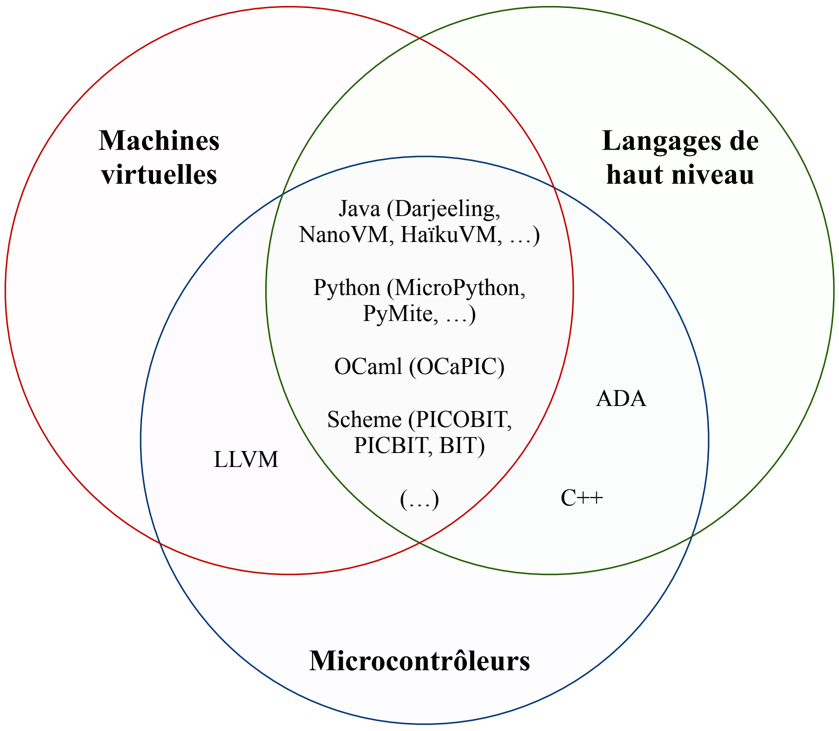

Machines virtuelles
Langages de haut niveau
Java (Darjeeling, NanoVM, HaïkuVM, …)
Python (MicroPython, PyMite, …)
OCaml (OCaPIC)
Scheme (PICOBIT, PICBIT, BIT)
(…)
ADA
LLVM
C++
Microcontrôleurs

FIGURE 1.7 – Microcontrollers, virtual machines, and high-level languages

hundred kilobytes of flash memory. Memory limitations can restrict the use of all aspects of a program. Moreover, most of these solutions are often not very portable : they tend to target a specific range of microcontrollers, and adapting them to a different architecture can be difficult.

In our work, we consider the approach adopted by OCaPIC to be the most suitable for programming microcontrollers. The OCaml virtual machine is lightweight enough to support execution of the full language on microcontrollers with quite limited memory resources, while the language itself provides significant advantages, such as greater program safety through static typing and automatic memory management. However, like most of the solutions discussed in this section, OCaPIC is quite limited in portability, as it is available only for PIC18 microcontrollers. In the next chapter, we therefore propose a *generic* OCaml virtual machine, designed to run on a wide range of targets, while maintaining a small memory footprint and satisfactory execution speed for a rich language.

1.3 Synchronous programming

The role of a microcontroller is to act as the *conductor* of an electronic circuit : it must react to stimuli from the electronic components to which it is connected (sensors, buttons, . . .) in order to send signals to other components of the circuit (LCD screens, actuators, . . .). For example, inside a computer keyboard, a microcontroller reacts in an imperceptibly short time to the various key presses made by the user. Programs for microcontrollers often require the hardware to react quickly to input signals, regardless of the order in which they occur : pressing certain keys on the keyboard should not, for instance, mask the simultaneous pressing of another key. Thus, microcontroller programming is inherently concurrent : the software components of an embedded program that handle signals from the system's environment must generally appear to react *at the same time*.

Language	Implementation	VM / Interpreter ?	Microcontroller (family)
Java	JVM	yes	-
	Darjeeling	yes	AVR, MSP
	NanoVM	yes	AVR
	HaikuVM	yes	AVR
	MicroEJ	yes	ARM
Python	CPython	yes	-
	MicroPython	yes	STM32
Scheme	Guile	yes	-
	BIT	yes	Motorola 68HC11
	PICBIT	yes	PIC18
	PICOBIT	yes	PIC18
OCaml	ZAM	yes	-
	OCaPIC	yes	PIC18
Ada	GNAT	no	STM32
	AVR-Ada	no	AVR
C++	avr-g++	no	AVR
	IAR	no	AVR, MSP, STM32
	MPLAB XC32++	no	PIC32, SAM
C, Go, Rust, . . .	LLVM	generally no (intermediate representation)	AVR, MSP

FIGURE 1.8 – Implementations of programming languages on microcontrollers

Embedded systems, sometimes safety-critical, therefore exhibit concurrent behaviors subject to more or less strict timing constraints. Such systems are often described as *real-time systems*.

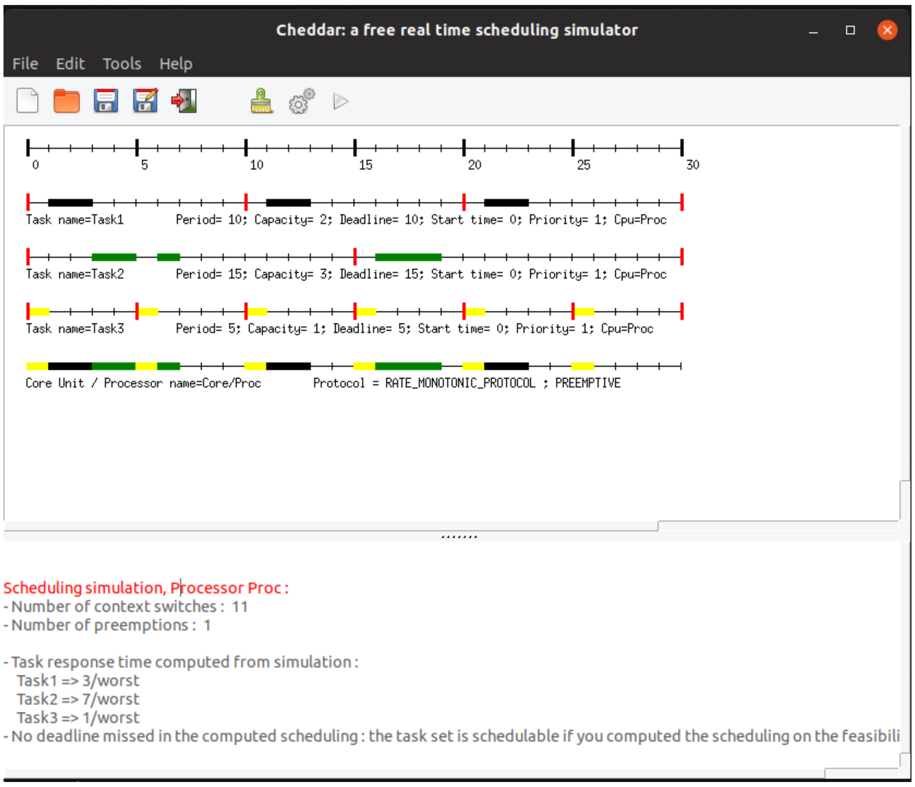
1.3.1 Real-time systems

A real-time system is a computer system in which the reaction and computation times of the different program components (called *tasks*) are subject to constraints that must be respected. These tasks handle the stimuli that the program is supposed to respond to, concurrently, as they occur. Such stimuli may arise during program execution either periodically or sporadically.

Building real-time systems introduces the notion of *scheduling*, which consists in defining the order in which the different tasks of a program will be executed (generally in a cyclic manner), taking into account the constraints associated with them (such as their frequency, priority, or deadline).

In the case of periodic tasks, program scheduling can be performed statically, prior to program execution : it is then possible to use software tools such as Cheddar [SLNM04] to automatically establish (if one exists) an execution order of the program's tasks that satisfies their timing constraints. Figure 1.9 shows an example of scheduling performed by Cheddar for three different periodic tasks. In a system containing only periodic tasks and without preemption, the execution of a program can simply correspond to sequential calls to the functions representing each task, for example by traversing a table that represents the scheduling sequence.

For sporadic tasks, which handle the occurrence of occasional events, scheduling cannot be performed « *offline* » (during program design), since the timing of such events is not predictable. Scheduling of the different tasks of a program is therefore carried out dynamically, as events that the program must react



```
Task name=Task1        Period= 10; Capacity= 2; Deadline= 10; Start time= 0; Priority= 1; Cpu=Proc
Task name=Task2        Period= 15; Capacity= 3; Deadline= 15; Start time= 0; Priority= 1; Cpu=Proc
Task name=Task3        Period= 5; Capacity= 1; Deadline= 5; Start time= 0; Priority= 1; Cpu=Proc
Core Unit / Processor name=Core/Proc        Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE

Scheduling simulation, Processor Proc :
- Number of context switches :  11
- Number of preemptions :  1

- Task response time computed from simulation :
   Task1 => 3/worst
   Task2 => 7/worst
   Task3 => 1/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibili
```

FIGURE 1.9 – Static scheduling of three periodic tasks with Cheddar

to occur. However, the schedulability of the system can still be guaranteed statically provided that the minimum interval between two occurrences of sporadic events of the same type is known.

A real-time program generally corresponds to a system that combines periodic stimuli with sporadic events. As a result, real-time systems make use of software *schedulers*, included in *real-time operating systems* (*RTOS*) such as FreeRTOS [GPPT16], which run alongside the program and dynamically assign control to tasks based, for instance, on their priority levels and the occurrence of events.

Static or dynamic scheduling methods, as well as the use of real-time operating systems, are relatively demanding and can be difficult to implement on certain devices : the memory resources required to run the software components responsible for scheduling and creating concurrent tasks are non-negligible, and may not be available on the microcontrollers we consider (which typically provide only a few kilobytes of memory). Moreover, even when a program has been proven schedulable, unforeseen errors may still occur due to concurrent access to resources shared by multiple tasks in a program, using synchronization primitives (mutual exclusion, synchronization barriers, etc.). This is the case, for example, with priority inversion phenomena, where a lower-priority task monopolizes a shared resource and never yields it to a higher-priority task. Such issues can have serious consequences : the program of *PathFinder*, NASA's space probe sent to Mars in 1997, encountered a priority inversion problem just a few days after landing, which caused several unexpected software resets [Jon97][4].

In the context of this thesis, we adopt a simpler concurrency model that can guarantee that such unforeseen behaviors cannot occur. This model, *synchronous programming*, has proven effective for programming safety-critical embedded systems (aircraft, nuclear power plants, . . .), and in our view represents a suitable solution for concurrent programming of microcontrollers [VVC16]. The synchronous

4. A patch sent from Earth was nevertheless able to fix the problem.

programming model allows concurrent aspects of a program to be represented without requiring the use of a software system responsible for task management. Indeed, synchronous programming can be regarded as a real-time programming model with *static* and *deterministic* scheduling, which does not rely on an embedded software scheduler. Nevertheless, it can also be used in contexts where schedulers are present, for example after compilation of synchronous components into periodic tasks executed on real-time platforms [PFB+11]. The lightweight nature of this approach—producing programs that can run on hardware without any operating system—makes it a paradigm particularly well-suited to programming resource-constrained microcontrollers.

1.3.2 The synchronous hypothesis

The simplicity and expressive power of the synchronous programming paradigm rest on a principle of abstraction known as the *synchronous hypothesis*, which states that the time taken by the different components of a program to compute output values from input values is considered to be zero. This hypothesis is comparable to abstractions used in circuit design (as noted in [BB91]) : the time required for an electrical signal to pass through a set of logic gates is generally ignored when designing circuits. Similarly, in Newtonian mechanics, the propagation speed of a gravitational field (i.e., the speed of light) is not considered, and interactions between bodies are treated as instantaneous.

In a synchronous program, the program's inputs are therefore assumed to be instantaneous with its outputs, and all instructions are considered to be executed within the same logical instant, called a *synchronous instant*. Of course, for this hypothesis to hold, either the interval between two inputs must be guaranteed to be sufficient, or a buffer must be used to ensure that no inputs are lost.

The abstraction introduced by the synchronous hypothesis is illustrated in Figure 1.10. It shows that the actual execution time of the code needed to process the inputs and produce the outputs (top of the figure) is abstracted away : the outputs o_n are produced « *at the same time* » as the arrival of the inputs i_n (bottom of the figure).

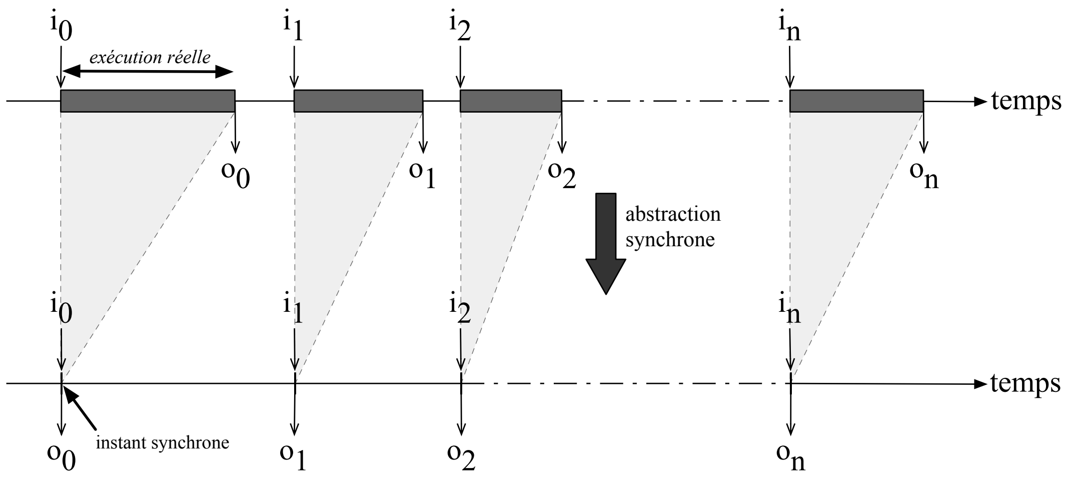


FIGURE 1.10 – The synchronous hypothesis :
The time required to compute output values (o_n) from input values (i_n) in a program is considered to be zero.

The execution of a synchronous program is a cyclic task that, at each instant, consists of reading the program's inputs (typically values coming from the system's environment) and computing output values to be emitted at the end of the instant. The synchronous hypothesis is satisfied as long as, at each instant, the interval between two program inputs is greater than the computation time of the output values. The program is thus synchronized with its inputs, and it can be considered that its reaction time is instantaneous.

The synchronous hypothesis thus simplifies reasoning about the concurrent aspects of a program by removing, from the developer's perspective, the temporal considerations involved in synchronizing the different software components of an application.

1.3.3 Esterel : an imperative synchronous programming language

The synchronous programming model emerged in the early 1980s, and one of the first languages to implement this model was created by a team of researchers at the École des Mines and INRIA Sophia-Antipolis. This language, called *Esterel* [BC84], is based on an imperative style in which the various components of a program communicate by broadcasting *signals*, potentially valued, that allow the exchange of information between concurrent components of a program. The Esterel language, whose *control-flow* semantics are driven by the occurrence of events during program execution, was initially designed with the aim of programming industrial robots at a high level of abstraction.

An Esterel program consists of several *modules*, each responsible for a specific task. The body of a module is imperative-style code containing standard operators such as sequencing (`;`) and conditionals (`if`), to which synchronous operators are added, including a parallel composition operator (`||`) as well as operators for emitting (`emit`) or waiting for (`await`) signals.

A common example of Esterel programming is shown in Figure 1.11. This example defines a module named `ABRO`, whose role is to wait, in parallel, for the presence of a signal `A` and a signal `B` before emitting a signal `O`. The presence of signal `R` resets the program's behavior.

```
module ABRO:

% Interface
input A, B, R;
output O;

% Body
loop
   [ await A || await B ];
   emit O
each R
end module
```

FIGURE 1.11 – Example of an Esterel program : the ABRO module

Beyond the programming of industrial embedded systems, Esterel's event-driven model is now used to program a variety of applications. For example, the programming language *ReactiveML* [MP05] is a synchronous reactive extension of OCaml that incorporates many aspects of the Esterel language. Like Esterel, it is based on a model of signal emission and reception, and it is designed for programming rich applications that take advantage of the concurrent features offered by the synchronous programming paradigm. ReactiveML programs span a wide range of uses, including musical applications, games, simulations of physical systems, and more classical algorithmic applications.

For instance, the ReactiveML program shown in Figure 1.12, taken from the official ReactiveML website [⚓21], performs a breadth-first traversal of a binary tree by concurrently executing the traversal of the tree's left and right subtrees.

```
(* Definition of binary trees. *)
type 'a tree =
  | Empty
  | Node of 'a * 'a tree * 'a tree

(* Breadth first traveral. *)
let rec process iter_breadth f t =
  match t with
  | Empty -> ()
  | Node (x, l, r) ->
      f x;
      pause;
      run (iter_breadth f l) || run (iter_breadth f r)
```

FIGURE 1.12 – Example of a ReactiveML program

Synchronous reactive programming is also well suited to applications in emerging domains, such as web application development. For example, the *Pendulum* language [SC16b] combines the algorithmic aspects of OCaml with a synchronous reactive model inspired by Esterel for the development of rich multimedia web applications. It relies on the *Js_of_OCaml* engine [VB14], which translates the bytecode of an OCaml program into a JavaScript application.

1.3.4 Lustre and Signal : declarative synchronous programming languages

Around the same time as the development of Esterel, other synchronous programming languages were created, though they were based on different models. In particular, the language *Lustre* [CPHP87] also emerged in the early 1980s, developed by researchers at the VERIMAG laboratory in Grenoble. Unlike Esterel, this synchronous programming language is not based on reacting to discrete events, but on a *data-flow* programming model, which represents the evolution of values over time. Lustre was originally designed to provide a programming language usable by control engineers accustomed to the declarative formalism of data-flow models [Hal05].

Following the model of the Lucid language [AW77][5], all values manipulated by a Lustre program are data flows : sequences of values that may vary during program execution. Each flow thus has, by default, a value at every instant of the program, and this value may change from one synchronous instant to the next. Consequently, a variable x in Lustre corresponds to the flow of all values taken by x instant by instant :

$$x \equiv (x_0, x_1, x_2, x_3, \dots, x_i \dots)$$

A constant therefore corresponds to an invariant flow of values :

5. LUSTRE was originally an acronym for « LUcid Synchrone Temps RéEl » (Lucid Real-Time Synchronous).

$$2 \equiv (2, 2, 2, 2, \dots, 2, \dots)$$

The arithmetic and logical operators of the language are applied pointwise to the values taken by the flows during program execution :

$$x + y \equiv (x_0 + y_0, x_1 + y_1, x_2 + y_2, x_3 + y_3, \dots, x_i + y_i, \dots)$$

Similar to Esterel modules, the basic software component of a Lustre program is the *node*. A node can be seen as a function that associates output flows with input flows. The body of a node is a system of equations, akin to temporal functions [CH86], which declare variables whose values may change at each execution instant. Each of these flows is computed within the same synchronous instant, and Lustre's synchronous data-flow model can be regarded as a restricted class of Kahn Process Networks (*KPN* [Kah74]), in which communications can be performed without *buffers* [MPP10].

The declarative style of a Lustre program is quite similar to the structure of a functional program : each equation defines a variable whose value depends on the current instant, and the system of equations forming the body of a node is essentially a set of variable declarations. Imperative considerations of program execution are absent from the semantics of the language : the reading order of the equations does not reflect their order of computation.

For example, Figure 1.13 defines a node named `BOOLOPS`, which takes as input two boolean flows, `A` and `B`, and produces as output the flows `ANDB`, `ORB`, and `XORB`, corresponding respectively to the computations $A \wedge B$, $A \vee B$, and $A \oplus B$.

```
node BOOLOPS (A:bool ; B:bool) returns (ANDB:bool; ORB:bool; XORB:bool);
let
  XORB = ORB and (not ANDB);
  ANDB = if A then B else false;
  ORB = if A then true else B;
tel;
```

FIGURE 1.13 – A Lustre node that computes the logical "and", "or", and "exclusive or" of its inputs

In the original version of Lustre, the available operators include the standard arithmetic and logical operators, as well as several *temporal* operators :

— The memory operator `pre`, which provides access to the value of a flow at the previous instant :

$$\texttt{pre}\ a \equiv (nil, a_0, a_1, a_2, \dots, a_{i-1}, \dots)$$

— The initialization operator `->`, which allows the definition of a flow by specifying one value for the first instant and another flow of values for the subsequent instants :

$$a\ \texttt{->}\ b \equiv (a_0, b_1, b_2, \dots, b_i, \dots)$$

For example, one can define in Lustre the sequence of positive integers as follows :

```
n = 0 -> pre n + 1
```

– Finally, the sampling operator `when`, which allows components of the program to be slowed down by conditioning the presence of flows on boolean values (the *clocks*) :

$$(a \texttt{ when } x)_n \equiv \begin{cases} a_n & \text{if } x_n = \texttt{true} \\ \perp & \text{otherwise} \end{cases}$$

Here, the symbol $\perp$ represents the absence of a value.

Each Lustre data flow is implicitly associated with a clock, which by default is the global (fastest) clock. It can be explicitly restricted to a slower clock using the `when` operator. For example, the following code snippet forces the flow `x` to be defined only when `b` is true :

```
x = 4 when b;
```

The table in Figure 1.14 shows a simulation of the execution of these three operators :

instant	0	1	2	3	4	5	...
c	true	false	true	true	false	true	...
x	x_0	x_1	x_2	x_3	x_4	x_5	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
`pre` y	*nil*	y_0	y_1	y_2	y_3	y_4	...
x `->` `pre` y	x_0	y_0	y_1	y_2	y_3	y_4	...
x `when` c	x_0	$\perp$	x_2	x_3	$\perp$	x_5	...

Figure 1.14 – The temporal operators of Lustre

The compilation of a Lustre program, known as *single-loop compilation*, consists in transforming the program into a sequential program containing a single loop. This loop polls the values of its input flows at the beginning of each synchronous instant, computes the values of its output flows, and emits these values at the end of the instant.

Signal

Based on a declarative model similar to Lustre, the language Signal [GG87] is a synchronous data-flow programming language designed for real-time systems. Developed in the 1980s by a team at the IRISA laboratory in Rennes, it was the third major synchronous language developed in France at that time.

Signal is a relational language that combines, in a sense, Lustre's data-flow perspective with Esterel's event-driven notion of signals. In Signal, a program defines signals (in an equational form similar to Lustre), which are viewed as sequences of data whose values are not necessarily present at every instant of program execution.

To represent such absence, Signal shares with Lustre the notion of a clock : the set of instants during which a signal has a value corresponds to its clock. Signal introduces the concept of synchronicity, which allows one to specify that two signals share the same clock and are therefore *synchronous*, using the operator `^=`.

Signal defines temporal operators that are quite similar to those of Lustre. For example, the `when` operator behaves like its Lustre counterpart, and the `$` operator, placed after the name of a signal, corresponds to Lustre's `pre`. For instance, one can define a counter `COUNT` that increments at each instant and is reset whenever a `RESET` event occurs, as follows :

```
(| COUNT := 0 when RESET default COUNT$ init 0 + 1 |)
```

The `default` operator allows one to define a value for `COUNT` when `RESET` is absent (it belongs to the so-called *polychronous* operators, since it applies to signals whose clocks may differ), while the `init` keyword specifies the initialization value of a signal at the first instant of execution.

Figure 1.15 shows a Signal process (equivalent to a Lustre node) named `COUNTERS`, which takes as input a signal (denoted by the symbol « ? ») named `RESET`, and returns two output signals (denoted by « ! ») called `CPT1` and `CPT2`. The first is an increasing counter, initialized to 0 ; the second is a decreasing counter, initialized to 100. Both can be reset whenever the `RESET` signal is present. The expression `CPT1 ^= CPT2` enforces that these two signals share the same clock.

```
process COUNTERS =
      ( ? event RESET;
        ! integer CPT1;
          integer CPT2;
      )
      (| CPT1 := 0 when RESET default CPT1$ init 0 + 1
       | CPT2 := 100 when RESET default CPT2$ init 100 - 1
       | CPT1 ^=CPT2
      |);
```

FIGURE 1.15 – A Signal process

Derived synchronous languages

A few years after the creation of Lustre, the core of the language was reused to define a set of industrial tools called *SCADE* [CPP17], which enables the graphical programming of synchronous systems by representing program components as sheets. This software suite is now widely used in critical applications, for example in the control systems of Airbus A300-series aircraft. Its adoption relies on a key feature of SCADE : the presence of a code generator, named KCG, which has received DO-178C Level A certification [⚓2], required for producing safety-critical embedded code intended for flight. This certification frees the development process from the obligation to carry out extensive testing to verify the correctness of the generated code, representing a significant advantage for the use of SCADE [PAM$^+$09].

This *factorization* of certification means that developers of an application only need to demonstrate traceability from the requirements specification to the SCADE program, rather than down to the executed code itself. Similarly, other KCG certifications allow SCADE to be used in the programming of systems with high safety requirements, such as the European EN 50128 standard for railway transportation, the international IEC 61508 standard for programmable components of electronic systems, and IEC 60880, which concerns the control systems of nuclear power plants.

Many academic synchronous languages, inspired by the data-flow model and in particular by Lustre (which itself continues to evolve in version 6), have been developed since its introduction. Among these

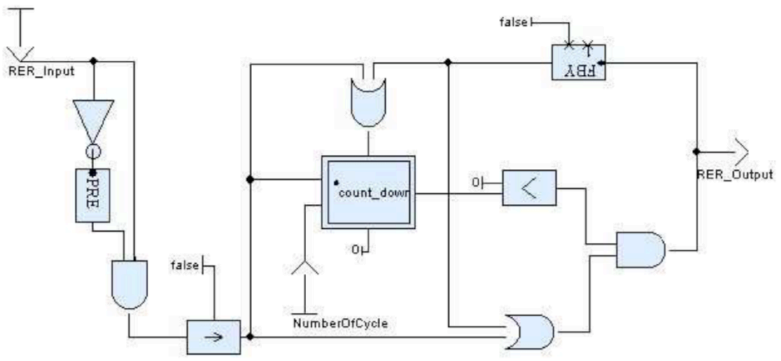


FIGURE 1.16 – A SCADE sheet
(image taken from [CHP06])

languages, Lucid Synchrone [CP99, Pou06] stands out as a higher-order extension of Lustre : in Lucid Synchrone, the parameters of nodes can themselves be synchronous nodes. Lucid Synchrone represents a powerful combination of Lustre-style synchronous languages and ML-style functional languages : for example, flows may correspond to values of product types, which can then be processed using pattern-matching operators. Several constructs from this language (signals, flow initialization analysis, state machines, clock systems, etc.) have since been integrated into SCADE.

```
let node iter init f x = y where
   rec y = f x (init -> pre y)
```

FIGURE 1.17 – Un nœud Lucid Synchrone qui itère une fonction `f` sur un flot de valeurs `x`

Other synchronous languages aim to extend the Lustre model. For example, the Heptagon language, developed by the PARKAS team at the École Normale Supérieure, includes an optimized representation of arrays of values [GGPP12], as well as an extension (called BZR) that integrates discrete controller synthesis (DCS) into the compilation of a synchronous program [DRM11]. The hybrid language Zélus [BP13], developed by the same team, derives from Lustre and Lucid Synchrone and combines notions of discrete time and continuous time through the use of ordinary differential equations.

In the context of our work, we consider the data-flow approach adopted by these synchronous languages to be particularly well suited to the hardware and applications we target. Indeed, the physical behavior of the hardware can be naturally represented by flows : each pin of a microcontroller has a value at any given moment (indicating whether or not it carries an electrical current), and each component of the application can thus be represented by a synchronous node that computes, at each instant, output current values from the electrical stimuli received as inputs by the microcontroller. The schematic representation used by the SCADE language, reminiscent of an electronic circuit diagram, clearly illustrates this close correspondence between the model adopted by a language such as Lustre and the underlying physical model. Moreover, Lustre's standard compilation model produces code that is resource-efficient and thus well adapted to the hardware limitations we consider.

1.4 Program Safety

Safety is the guarantee that a particular undesirable event cannot occur [Lam77]. More specifically, the safety of a program refers to the assurance that an abnormal or unexpected behavior cannot arise during its execution [AS87]. Such guarantees may concern various aspects of program behavior — for example, ensuring that protected memory regions are not accessed during execution [ACR+08], or that errors related to the illegal dereferencing of null pointers cannot occur while a program is running. In this sense, safety guarantees allow a developer to ensure that their program cannot reach states that fail to satisfy a desired property (such as the absence of *runtime errors*).

These guarantees are especially valuable in the programming of embedded systems, where undesirable program behavior can lead to disastrous consequences [WDS+10]. A critical embedded system that reacts unpredictably may cause serious accidents, which must be prevented at all costs. Yet the low-level programming models typically used for microcontroller development offer few guarantees to ensure the consistency of program behavior. Errors that could have been detected statically may go unnoticed at compile time in traditional languages. For instance, the weak type-checking of the C language permits incongruous, often unintended operations, such as multiplying an integer by the value of a character.

The use of higher-level programming languages often makes it possible to detect, at compile time, many errors that might otherwise appear during execution, thereby rejecting incorrect programs before they run (even if this sometimes results in rejecting certain programs that would have been correct). In this manuscript, we focus in particular on guarantees related to type safety, the calculation of the worst-case execution time of programs, and methods for formally specifying a language and developing its associated metatheory.

1.4.1 Static Typing

Many high-level programming languages, such as Haskell, Java, or OCaml, implement a mechanism of *static typing*. This consists in associating a type (that is, information about the nature of the values carried by a variable or object) with each identifier at compile time [Pie02]. This mechanism makes it possible to check, before program execution, for errors arising from the use of operators or functions that are incompatible with the types of the data to which they are applied. Static typing therefore increases type safety by rejecting incorrect programs during compilation.

As a result, the use of a statically typed programming language such as OCaml is a significant advantage for programming embedded systems, which may sometimes be safety-critical : the additional detection of typing errors at compile time helps prevent inappropriate physical reactions in programs, such as powering dangerous electronic components (for example, heating resistors, which can exceed 200 degrees Celsius when switched on). In the next chapter, in Section 2.3.3, we will see an example that uses advanced typing features (through GADTs — *Generalized Algebraic Data Types*) to represent low-level interactions by leveraging OCaml's expressive static type system.

Similarly, the type systems of domain-specific languages, such as Lustre, can further enhance program safety. Indeed, the synchronous clock type system of Lustre and its derivatives makes it possible to explicitly state the conditions governing the presence of certain values, and thus statically prevents attempts to read a value from an *absent* flow. Furthermore, the higher level of abstraction provided by synchronous programming languages allows additional guarantees to be checked during their various compilation phases — for example, verifying the *causality* of a program, which ensures that the concurrent components executed by the program can be statically sequentialized [BCE+03].

1.4.2 Worst-Case Execution Time

The synchronous hypothesis, on which Lustre and its derivatives are based, is only valid if the time required to compute a program's outputs is always shorter than the interval between the appearance (or sampling) of its inputs. In the context of strict real-time systems, it is therefore essential, in order to meet deadlines, to ensure that the time needed for the program to compute output values from arbitrary input values never exceeds, in the worst case (i.e., when execution is slowest), a maximum time bound. This *Worst-Case Execution Time* (WCET) must be strictly less than the shortest period separating two consecutive inputs, or the minimum delay between two sporadic events of the same type, so that no input is missed during program execution.

For a long time, the most common practice for estimating a program's maximum execution time relied on empirical measurements : the program was executed multiple times on the target hardware, and the longest observed runtime was taken as a realistic upper bound for its execution time [WEE+08]. However, this approach has clear limitations : since test coverage can never be exhaustive, it is likely that in real-world execution a specific combination of input values could trigger a longer reaction time than the one observed during testing. Safer techniques now exist to compute WCET by determining the maximum number of machine cycles required for execution. These techniques generally rely on static analyses that generate a *control-flow graph* representing the program's execution. The graph is then analyzed to identify all *valid* execution paths of the program and to estimate the cost of the longest valid path. Several static analysis methods exist for this purpose, such as the Implicit Path Enumeration Technique (IPET) [LM97]. IPET encodes program components as integer linear constraints, which are then used to compute the WCET as a linear expression to be maximized. This maximization can be carried out using Integer Linear Programming (ILP) solvers or constraint programming techniques. Several tools implement this technique, such as OTAWA [BCRS10], developed at the Toulouse Institute of Computer Science Research (IRIT), which estimates the WCET of programs on a wide range of architectures.

Other WCET estimation methods rely on analyzing the structure of a program's source code : the program's syntax tree is traversed in order to group together, in the control-flow graph, multiple nodes corresponding to syntactic constructs (loops, conditionals, etc.), and this process is repeated until the overall cost of the program can be deduced. For example, the Heptane tool [CP01] performs such an analysis and estimates the WCET of programs executed on ARM and MIPS architectures.

Figure 1.18 illustrates the difference between : (i) the worst-case time estimated by repeated measurements (which may underestimate the true WCET), (ii) the bound provided by WCET estimation tools such as Heptane or OTAWA (which may conversely overestimate the duration), and (iii) the actual worst-case execution time.

The computation of a program's WCET can be greatly complicated by the architecture of the processor on which it runs. Modern processors typically include multiple cache levels and advanced optimizations (pipelines, branch prediction, etc.), which make the execution time of an instruction variable and dependent on the execution history of preceding instructions. As a result, on such architectures it is not possible to calculate the WCET of a program by considering each instruction's cost independently : instead, an abstract model of the processor's behavior must be used to estimate WCET.

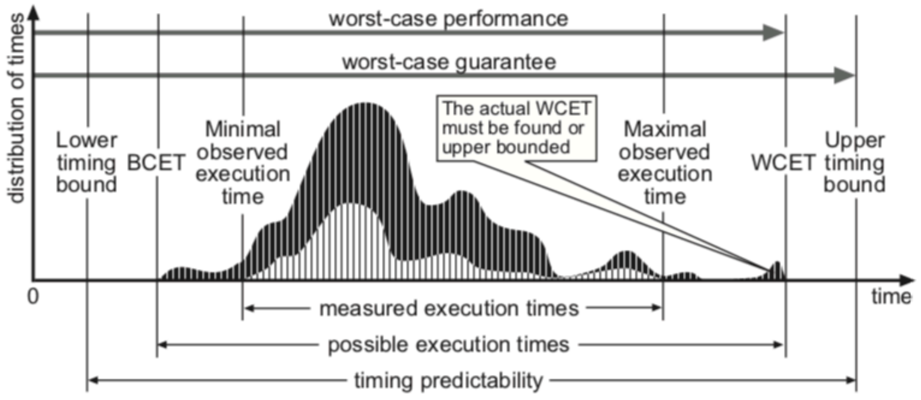


FIGURE 1.18 – Measured, guaranteed, and actual worst-case execution times (image from [WEE+08])

However, the microcontrollers we target here do not implement these sophisticated mechanisms that make WCET estimation so complex. For example, a microcontroller in the ATmega328 family has no cache system, its processor does not implement branch prediction, and it is limited to a simple two-stage scalar pipeline that merely fetches the next instruction while the current one is being executed. This limitation significantly simplifies the computation of the worst-case execution time, since the execution time of each instruction can be analyzed separately.

In particular, in Chapter 6 we will present a method for computing the WCET of a program running on a microcontroller by analyzing the bytecode instructions of which it is composed.

The estimation of WCET for synchronous programs is usually carried out on the sequential program generated by compilation. In the case of Lustre, this program takes the form of a single loop, without recursion or dynamic allocation. The simplicity of the generated code thus makes it straightforward to bound the execution time of a loop iteration using automated software solutions.

1.4.3 Formal Specifications and Metatheory

A formal specification defines, in an unambiguous language, the rules governing the behavior of a system [Spi89]. In the context of programming languages, such rules describe, for example, the appropriate type systems or the semantics of the language.

Deep specification tools [ABC+17], such as the proof assistants Coq [Tea19] and Isabelle/HOL [NPW02], as well as formal methods such as the B-Method [Abr96] — widely used in critical industrial applications (for instance, in the automation of Paris Metro lines 1 and 14) — enable the formal specification of systems in order to certify their correctness. From such formal specifications, the above tools can generate executable code in a general-purpose programming language : for example, Coq allows the extraction of functions to OCaml (among other languages), while Isabelle/HOL supports code generation to Haskell and other high-level languages. Through successive refinement of the specification, it is possible to obtain certified-correct executable programs. Prominent examples include the seL4 microkernel [KEH+09], whose implementation correctness was proven using Isabelle/HOL ; the CompCert certified compiler [KLW14], verified in Coq, which supports C compilation for numerous processor architectures (ARM, RISC-V, x86, etc.) ; the CakeML project [KMNO14], which features a formally specified and verified compiler (using the HOL4 proof assistant [SN08]) for a functional programming language ; and JSCert

[BCF+14], a Coq formalization of JavaScript, together with its interpreter JSRef, which has been proven to conform to the formal specification.

The Lustre language was designed from the outset with a formally defined semantics, illustrating the intent of its creators to target systems that require strong safety guarantees. Many research efforts have focused on the formalization of Lustre and the development of certified compilers ensuring that generated programs respect its semantics [BBD+17, BBP18, Aug13]. These works involve formalizing a Lustre-like language and certifying, using Coq, a code generator compatible with CompCert, thereby yielding a fully certified compilation chain from Lustre programs down to machine code.

Chapter Conclusion

The main purpose of this thesis is to bring together all the aspects presented in this chapter. Indeed, because of the advantages offered by high-level languages, we aim to execute programs written in such a language on microcontrollers with very limited hardware resources. The use of the OCaml language appears to us to be a relevant approach, both for the richness of the language, the lightweight nature of its model, and the safety it provides through its strict static typing, which ensures that no type-related errors will occur during program execution.

Thus, in the following chapter we will describe a portable implementation of the OCaml virtual machine, named OMicroB. This virtual machine is designed to run on a wide variety of microcontrollers, in some cases with less than 4 kilobytes of RAM. Since applications for microcontrollers are inherently concurrent due to their multiple interactions with their environment, we will then propose OCaLustre, a synchronous extension of the OCaml language, compatible with the aforementioned virtual machine. This extension will bridge the gap between the advantages of using a high-level general-purpose language and a lightweight concurrency model suited to microcontroller resources.

Our solutions will benefit from the guarantees offered by both worlds, and static analyses may furthermore be carried out directly on the bytecode of an OCaml program, thereby providing an increased level of abstraction for developing applications that leverage the portability of this approach. In this regard, we will describe an analysis for computing the worst-case execution time of a synchronous instant in an OCaLustre program, ensuring that activations do not overlap and thereby validating the synchronous hypothesis.

Moreover, while proving the correctness of a full compilation chain for a programming language lies beyond the scope of this thesis, we will nevertheless address many formal aspects in the description of OCaLustre, for which we will establish a formal specification. Certain metatheoretical properties will also be proven using the Coq proof assistant.

2 OMicroB : A Generic OCaml Virtual Machine

The work initiated by the OCaPIC virtual machine appears to constitute a promising and well-suited solution for programming microcontrollers with limited resources. Indeed, the use of the OCaml language provides microcontroller programming techniques with a modern development language, which supports multiple paradigms such as functional programming, object-oriented programming, and, of course, the more traditional imperative programming model. Safer than the conventional assembler/C pair, this high-level language enables the early detection of certain errors in programs through its static type system. Furthermore, OCaml offers tools that simplify and guarantee program development, for instance through its automatic memory management system (*garbage collection*) or its type inference mechanism, which allows developers to avoid writing a large number of superfluous type annotations while still preserving the guarantees of static typing.

Nevertheless, OCaPIC is not well-suited to the variety of microcontroller models available on the market. The decision to implement this tool largely in the assembly language of PIC18 microcontrollers yields significant performance (in terms of execution speed), but greatly limits the portability of this virtual machine, since it can only be executed by hardware from that family. Given the diversity of microcontroller families and architectures used by both industry and hobbyists (AVR, PIC16, PIC32, ARM, . . .), we consider that a *generic and portable* solution, capable of running on a wide range of targets, would represent a major advantage for the adoption of high-level programming methods in the field of microcontroller development.

We therefore propose **OMicroB**, a *generic* virtual machine designed to execute OCaml programs on diverse and resource-constrained hardware. This virtual machine, developed in collaboration with Benoît Vaugon (the designer of OCaPIC), takes advantage of the ubiquity of the C programming language in traditional microcontroller development, which ensures the almost systematic availability of C compilers for every target platform. Accordingly, a large part of the OMicroB virtual machine is written in C, which we use as a form of *portable assembly language*, allowing simplified porting of OMicroB to otherwise heterogeneous devices.

Given the stringent constraints on the memory sizes of the devices considered, we have sought to provide a finely controlled implementation that remains executable on hardware with very limited resources. Particular effort has therefore been devoted to reducing the memory footprint of programs through a variety of optimizations.

This chapter first recalls the main syntactic and semantic aspects of the OCaml programming language, before describing in detail the implementation of the standard OCaml virtual machine, and finally presenting our implementation of a virtual machine designed to run on diverse devices with limited resources.

2.1 The OCaml Language

This section is intended to familiarize readers who are not accustomed to OCaml with the syntax and essential features of the language. We present here the main aspects of the language that will be used in the examples throughout this dissertation. Readers already familiar with OCaml may skip directly to the next section. What follows is a non-exhaustive overview of the main high-level programming constructs that allow developers to build expressive applications that are both rich and safer, thanks to OCaml's static type system and its automatic memory management mechanism.

Functional values : The core of the OCaml language is both functional and imperative. It allows functions defined in the language to be manipulated as values. For example, the function `compose` computes the application of the composition $(f \circ g)$ to a variable x :

```
let compose f g x = f (g x)
```

This notation is an alternative to a more explicit form, which defines a function using the keyword **fun**. The definition of the function `compose` can therefore also be written as follows :

```
let compose = (fun f g x -> f (g x))
```

The same function can also be expressed in a *curried* form. *Currying* consists in transforming a function of n arguments into a function that takes a single argument and returns a chain of $n-1$ functions, each of which also takes a single argument, until finally returning the value computed by the body of the original function [Rey98]. Thus, the following definition of the function `compose` is semantically identical to the two previous definitions :

```
let compose = (fun f -> (fun g -> fun x -> f (g x)))
```

OCaml also supports the use of *lambda-expressions*, i.e. anonymous functions that can be employed inside the body of other functions. For example, using the function `compose`, the following function applies an anonymous function (defined with the same keyword **fun**) that doubles its parameter x, and then prints the result (via the primitive `print_int`) :

```
let double_print x = compose print_int (fun x -> x * 2)
```

It should be noted that functional values manipulated in programs, also known as *closures*, may capture and retain their own environment. For instance, the function `return_closure` below computes, from a given x, a closure that stores the value of x in its environment and awaits a parameter y in order to compute the result $x + y$.

```
let return_closure x = (fun y -> x + y)
```

The type of this function is a *function type* of the form *type of the parameter* → *type of the result*. Since `return_closure` takes an integer as input and returns a function that itself takes another integer and produces an integer (because the operator + applies only to integers and produces an integer[1]), its type is :

$$int \rightarrow (int \rightarrow int)$$

The function types in OCaml are right-associative, which allows the more concise notation :

$$int \rightarrow int \rightarrow int$$

Tuples : A function can manipulate tuples of values. For example, the function `sum_triple` computes the sum of the three elements contained in the triple passed as a parameter :

```
let sum_triple (x,y,z) = x + y + z
```

The function `sum_diff`, on the other hand, returns a pair corresponding to the sum and the difference of the values passed as parameters :

```
let sum_diff x y = (x + y, x - y)
```

Static typing, polymorphism, and type inference at compilation : In OCaml, the definition of variables and functions does not generally require the developer to annotate each definition with its type. The types of values are inferred by the compiler, which then statically checks that the implemented programs are consistent with the types deduced during a *type-checking* phase at compilation.

For example, the type of the function `sum_triple`, inferred by the compiler, is $(int * int * int) \rightarrow int$[2] because the three elements of the input triple are added together with the integer addition operator « + ». Similarly, the compiler infers for the function `sum_diff` the type $int \rightarrow int \rightarrow (int * int)$. Any use of these functions with values incompatible with their inferred types (for instance, passing a triple of floating-point values to `sum_triple`, or attempting to use the result of `sum_diff` as if it were a plain integer) will be detected and rejected at compile time.

The strict static typing of the language thus increases program safety by ensuring that no error due to inconsistent typing of values can occur during execution. Type inference also spares the developer from having to annotate values with their most general type. The notion of a *most general type* stems from the fact that OCaml implements a type system with parametric polymorphism, which allows functions to be defined generically so that they can apply to arguments of various types. For instance, the following `identity` function simply returns its argument x :

```
let identity x = x
```

The type of this function, as inferred by OCaml, is $\alpha \rightarrow \alpha$ (also written `'a -> 'a`), meaning that the function receives an argument of an arbitrary type (α) and returns an argument of the same type.

1. It is the operator « `+.` » that adds two floating-point numbers.
2. The symbol $*$ in the type of a tuple separates the types of its individual elements.

In the same way, the compiler infers for the `compose` function, introduced at the beginning of this section, the following type :

$$(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$$

Algebraic data types and pattern matching : An OCaml program can define new data types in the form of *algebraic data types* (which generalize enumerated types). Defining such a type consists of assigning it a name along with several constructors.

For example, the enumerated type `suit` can represent the suit of a playing card :

```
type suit = Spade | Heart | Diamond | Club
```

It is also possible to define parameterized constructors, which are particularly useful to represent recursive types. For example, the type `suit_list` represents a list of card suits, either the empty list (`Nil`), or a list constructed (with the constructor `Cons`) from one suit (the head of the list) and another list of suits (its tail) :

```
type suit_list = Nil | Cons of suit * suit_list

(* the list [Heart; Club; Club] : *)
let example_list = Cons(Heart, Cons(Club, Cons(Club, Nil)))
```

To manipulate the values of an algebraic type, one typically uses *pattern matching* over its constructors, with the **`match ... with`** construct. For example, the recursive function `length` computes the length of a list of suits :

```
let rec length l =
  match l with
  | Nil -> 0
  | Cons (_, l') -> 1 + length l'

let l = Cons(Heart, Cons(Club, Cons(Club, Nil))) in
  print_int (length l)  (* prints "3" *)
```

Note that in a pattern, the character _ denotes « any value ». For example, in the pattern `Cons (_, l')`, used in the **`match`** of the function `length`, the first element of the list is ignored since its value is irrelevant to the computation, and does not need to be named.

A type can also be defined by exploiting the mechanism of parametric polymorphism. For example, the type that represents a list whose elements are of any type can be defined as follows :

```
type 'a list = Nil | Cons of 'a * 'a list

(* the list [Heart; Diamond; Diamond] of type "suit list" *)
let suit_list_example = Cons(Heart, Cons(Diamond, Cons(Diamond, Nil)))

(* the list [1; 2; 3; 4] of type "int list" *)
let int_list_example = Cons(1, Cons(2, Cons(3, Cons(4, Nil))))

(* the list [2.33; 4.55; 6.77] of type "float list" *)
let float_list_example = Cons(2.33, Cons(4.55, Cons(6.77, Nil)))
```

With such a definition it becomes possible, among other things, to create lists of integers, floats, values of a sum type, functions (all of the same type), or even lists of lists. However, lists must remain homogeneous : the type `'a list` defines the structure of a list containing values of a certain type, but all the elements of the *same* list must share the *same* type.

Furthermore, this polymorphic list type `'a list` is predefined by OCaml's standard library. The constructor `Nil` is written `[]` and the constructor `Cons` corresponds to the infix operator `::`. Thus, the list `Cons(1, Cons(2, Cons(3, Nil)))` can be written `1::2::3::[]`, or even more simply `[1;2;3]` thanks to syntactic sugar provided by the compiler.

Record types : Values composed of several distinct elements can be represented using *record types* (or *product types*). A record type is defined by giving the name and type of each element contained in the value.

The following example defines a record type to represent a point in a plane (with two coordinates : x and y), as well as a function that computes the distance between two points `p1` and `p2` :

```
type point = { x : float ; y : float }

let distance p1 p2 =
  let x_diff = p1.x -. p2.x in
  let y_diff = p1.y -. p2.y in
  sqrt (x_diff *. x_diff +. y_diff *. y_diff)
```

Mutability and imperative programming : The contents of the fields of a record type can be declared as mutable during program execution using the keyword **`mutable`**. The value of a mutable field can be modified with the operator `<-`.

For example, the code below declares a point whose fields are mutable, and a function that translates a point p by a vector with coordinates (u, v). This translation modifies in place the values contained in the fields x and y of p (the infix operator `;` denotes a sequence of actions) :

```
type point = { mutable x : float ; mutable y : float }

let translate p (u,v) =
  p.x <- p.x +. u;
  p.y <- p.y +. v
```

In general, as in many functional languages (like Haskell or Scheme), variables in OCaml are immutable : once declared, their value cannot be modified during execution.

However, a special data type called a *reference* allows variables whose value can change during execution, making imperative programming possible. A reference is defined with the keyword **`ref`**, its contents accessed with the operator `!`, and updated with the operator `:=`.

The following program defines a counter `count`, used to run a loop 10 times. In each iteration, the value of `count` is printed, then incremented :

```
let loop =
  let count = ref 0 in
  while (!count < 10) do
    print_int !count;
    count := !count + 1
  done
```

In fact, the type of a reference corresponds to a record with a single field named `content`, declared as *mutable* :

```
type 'a ref = { mutable content : 'a }
```

Other basic data structures in OCaml are also mutable, such as arrays[3].

For example, the following program uses a « for » loop (as in many programming languages) to add 2 to each element of the array *t* (defined with the syntax `[| ... |]`). Each element *i* of array *t* is accessed with the notation `t.(i)` :

```
let t = [| 1 ; 2 ; 3 ; 4 ; 5 |] in
for i = 0 to 4 do
  t.(i) <- t.(i) + 2
done
```

Exceptions : As in many programming languages, OCaml provides an exception mechanism, allowing the control flow of a program to change when an error occurs.

For example, the program below defines an exception `Division_by_zero`, which is raised in the function `divide` if its second parameter is equal to `0.0`. This exception is then caught using the construct **`try _ with`** in the calling function `call_divide`, which returns instead the predefined value `max_float` :

3. Historically, strings were mutable too, but this behavior is now deprecated. Byte sequences can be used instead, via the *Bytes* module.

```
exception Division_by_zero

let divide x y =
  if y = 0.0 then raise Division_by_zero;
  x /. y

let call_divide x y =
  try divide x y
  with Division_by_zero -> max_float
```

Modules : OCaml is a *modular* language, which makes it possible to group together a set of type and value declarations (variables, functions) inside distinct modules. For example, the following *Card* module defines a set of types and functions for manipulating playing cards.

```
module Card = struct

type suit = Spade | Heart | Diamond | Club
type rank = Number of int | Jack | Queen | King | Ace

(* A card = a rank and a suit: *)
type card = { r : rank ; s : suit }

(* Card values for the game of Belote: *)
let score card trump =
    match card.r with
    | Number 10 -> 10
    | Number 9 -> if card.s = trump then 14 else 0
    | Number _ -> 0
    | Jack -> if card.s = trump then 20 else 2
    | Queen -> 3
    | King -> 4
    | Ace -> 11

(* Display function: *)
let print_card card =
   let suit_string =
      match card.s with
        | Heart -> "Heart"
        | Spade -> "Spade"
        | Club -> "Club"
        | Diamond -> "Diamond"
   in
   let rank_string =
     match card.r with
     | Number i -> string_of_int i
```

```
      | Jack -> "Jack"
      | Queen -> "Queen"
      | King -> "King"
      | Ace -> "Ace"
    in
    print_string (rank_string ^ "␣of␣" ^ suit_string)
end
```

Any file `foo.ml` implicitly corresponds to the declaration of a module `Foo`, whose interface can be defined in the file `foo.mli`. This modular programming model makes it possible to perform the separate compilation of the distinct components of a program.

Objects : Finally, OCaml is also an object-oriented programming language. The object type system implemented in the language supports multiple inheritance as well as class polymorphism. For example, the following OCaml code (taken from the official OCaml website [⚓29]) defines a class representing a polymorphic stack structure.

```
class ['a] stack =
    object (self)
      val mutable list = ( [] : 'a list )  (* variable d'instance *)
      method push x =
        list <- x :: list
      method pop =
        let result = List.hd list in
        list <- List.tl list;
        result
      method peek =
        List.hd list
      method size =
        List.length list
    end
```

Due to the use of parametric polymorphism (denoted by the parameter `'a` of the class), the way an instance of this class is used determines the type of the values stored in the stack it represents. For example, the following program manipulates a stack of playing cards :

```
open Card

let () =
  let s = new stack in
  s#push {v = Jack ; c = Spade};   (* s is therefore a ''card stack'' *)
  s#push {v = Queen ; c = Heart};
  let c = s#pop in
  print_card c;   (* prints ''Queen of Heart'' *)
  print_int s#size (* prints 1 *)
```

Automatic memory management : As in the Java language, OCaml implements a *garbage collector*. This mechanism automatically frees the memory associated with values that are no longer used by the program at runtime. Thus, using the OCaml language spares the developer from concerns related to the dynamic deallocation of the various values being manipulated. Moreover, this automation makes programs less prone to bugs caused by programmer errors in handling program memory, which are often quite difficult to detect.

Advanced constructions : Finally, OCaml implements several other high-level programming features, such as GADTs (*Generalized Algebraic Data Types*), *polymorphic variants*, or *parameterized modules* (also called *functors*). These advanced constructions are fully compatible with the generic virtual machine that we describe in this chapter. We will present some examples making use of such constructions in the course of our discussion (for instance, in Section 2.3.3 we will present an example using GADTs to enhance the type safety of primitives that implement communication between a microcontroller and its environment).

2.2 The ZAM : Reference OCaml Virtual Machine

OMicroB is a virtual machine derived from the standard OCaml virtual machine. As such, it shares many common features with the latter. In this section, we present the main technical aspects of the reference OCaml virtual machine, from the format of the *bytecode* it can interpret, to a brief description of its runtime library, including the details of how OCaml values are represented internally.

2.2.1 Overview of the ZAM

Developed since 1996 at Inria within several research projects (*Cristal*, then *Gallium*), this virtual machine is a stack-based machine, with a functional and imperative core, which can be seen as a version of the Krivine abstract machine [Kri07] implementing a strict application model rather than call-by-name. Also known as the ZAM (*Zinc Abstract Machine*) in reference to the ZINC project [Ler90] from which it originated[4], the reference OCaml virtual machine constitutes one of the two compilation targets of the OCaml language. Indeed, an OCaml program can be compiled either into a *bytecode* file (via the *ocamlc* compiler), interpretable by the ZAM (more precisely by its reference implementation, named *ocamlrun*), or into a native executable via the *ocamlopt* compiler. In this thesis, we focus on the virtual machine approach to OCaml compilation, due to the portability opportunities it offers, as well as the relative simplicity and lightweight nature of this compilation model for the OCaml language.

2.2.2 OCaml Bytecode

In version 4.06, the standard OCaml virtual machine contains a set of 148 different bytecode instructions, into which any OCaml program can be compiled. These bytecode instructions correspond to operations that act on the control flow of programs (such as the conditional branching instruction *BRANCHIF*), instructions that perform arithmetic or boolean computations (such as the *MULINT* instruction, which multiplies integer values), instructions for dynamic memory allocation (such as the *CLOSURE* instruction, which stores a functional value in the form of a closure in the virtual machine's heap), or

4. This project aimed at a lightweight implementation of the ML language.

instructions that manipulate the virtual machine's stack (such as the *PUSH* and *POP* instructions, which push and pop values).

Most bytecode instructions are in fact shortcuts corresponding to combinations of several atomic instructions (for example, the instruction *PUSHACC1* has the same behavior as the instruction *PUSH* followed by the instruction *ACC1*, which retrieves the second element on the stack).

For the sake of brevity, the complete set of OCaml bytecode instructions and their descriptions are not reproduced in this thesis ; however, the main instructions will be presented[5].

The generation of OCaml bytecode from an OCaml source file is performed by the *ocamlc* compiler. The file produced by *ocamlc* is an executable program that contains several distinct sections, each necessary for the initialization and interpretation of the OCaml program by the virtual machine :

— A CODE section containing the bytecode instructions corresponding to the compiled program code.
— A DATA section containing a serialized representation of the program's global variables (constants, exceptions, etc.).
— A PRIM section, which is a table mapping the external primitives used by a program to the integers that serve as references to them in the bytecode.
— An optional DLLS section containing the names of external libraries required for program execution (for example, the `Unix` library, which provides access to system calls).
— An optional DLPT section containing the paths of the libraries used by the program.
— An optional DBUG section, used for program debugging.

For example, Figure 2.1 shows the OCaml definition of a function `facto`, which computes the factorial of an integer, together with its application to the value 4. A representation of the bytecode contained in the file generated by *ocamlc* is reproduced alongside it. This bytecode is only a fragment of the CODE section of the file, which, in addition to the instructions corresponding to the `facto` function, also includes a large number of bytecode instructions required for program initialization. Moreover, the generated bytecode also contains the code of functions from a module imported by default into every OCaml program, named `Pervasives`, which provides the definitions of standard functions (for instance, printing functions) and basic operators (such as integer addition or logical "or").

2.2.3 Structure of the Virtual Machine

The OCaml virtual machine manipulates, during the interpretation of a program, several registers required for its execution. These registers are as follows :

— An accumulator (`acc`), used to store a value in order to avoid excessive pushes and pops on the virtual machine stack.
— A pointer (`pc`) to the next bytecode instruction to be interpreted.
— A pointer (`sp`) to the topmost cell of the stack.
— A pointer (`trapSp`) to the current exception handler.
— A counter (`extra_args`) representing the number of arguments still to be applied to a function.
— A pointer to the environment (`env`) of the current closure.
— A pointer to the table of global variables (`global_data`) of the program.

5. Readers interested in the semantics of each bytecode instruction may refer to the documentation of the Cadmium project [⚓6].

```
let rec facto x =
  match x with
  | 0 -> 1
  | _ -> x * facto (x-1)
in
facto 4
```

```
(...)
1673   BRANCH 1685

1674   ACC 0
1675   BRANCHIFNOT 1683
1676   ACC 0
1677   OFFSETINT -1
1678   PUSHOFFSETCLOSURE 0
1679   APPLY 1
1680   PUSHACC 1
1681   MULINT
1682   RETURN 1
1683   CONST 1
1684   RETURN 1

1685   CLOSUREREC 1 0 1674 []
1686   CONST 4
1687   PUSHACC 1
1688   APPLY 1
1689   POP 1
```

FIGURE 2.1 – Definition and application of the factorial function, and corresponding bytecode

The contents of these registers are updated during program execution according to the semantics of each bytecode instruction contained in the file generated by *ocamlc*. The OCaml bytecode interpreter of the virtual machine is implemented in the C programming language.

Figure 2.2 illustrates the effect of each instruction of the bytecode program presented in Figure 2.1 on the state of the virtual machine registers.

Position	Instruction	Description
1673	`BRANCH 1685`	Jump to instruction 1685 (`pc = 1685`)
1674	`ACC 0`	Load the top of the stack into the accumulator (`acc = sp[0]`)
1675	`BRANCHIFNOT 1683`	If `acc` = 0, then jump to instruction 1683 (`pc = 1683`)
1676	`ACC 0`	Load the top of the stack into the accumulator (`acc = sp[0]`)
1677	`OFFSETINT -1`	Decrement `acc`
1678	`PUSHOFFSETCLOSURE 0`	Push `acc`, and load into `acc` the closure stored in `env`
1679	`APPLY 1`	Push the current context (`extra_args`, `pc`, `env`), put `acc` into `env`, and jump to the code of the closure in `acc` (i.e. `facto`)
1680	`PUSHACC 1`	Push `acc`, and load `sp[1]` into `acc`
1681	`MULINT`	Multiply `acc` and `sp[0]`, store the result in `acc`, and pop `sp[0]`
1682	`RETURN 1`	Pop one element and return from the function (`pc = pop(), env = pop(), extra_args = pop()`)
1683	`CONST 1`	Load constant 1 into `acc`
1684	`RETURN 1`	Return from the function (`pc = pop(), env = pop(), extra_args = pop()`)
1685	`CLOSUREREC 1 0 1674 []`	Create the closure whose code starts at address 1674 (i.e. `facto`) in the accumulator, and push it onto the stack
1686	`CONST 4`	Load the constant 4 into `acc`
1687	`PUSHACC 1`	Push `acc`, and load `sp[1]` into `acc`
1688	`APPLY 1`	Push the current context (`extra_args`, `pc`, `env`), put the value of `acc` into `env`, and jump to the code of the closure in `acc` (i.e. `facto`)
1689	`POP 1`	Pop one value (`sp--`)

FIGURE 2.2 – Interpretation of the bytecode for the factorial program of Figure 2.1

2.2.4 Representation of Values

Due to the parametric polymorphism of the OCaml language, all values manipulated by the ZAM are based on a uniform representation : OCaml values all have the same length, defined by the compiler depending on the architecture of the machine on which the program is executed (typically 32 or 64 bits). Nevertheless, OCaml's garbage collector in particular needs, at runtime, to distinguish between immediate values, which it should ignore, and dynamically allocated values on the heap of the virtual machine, which it must visit and process.

This dichotomy in the ZAM is very simple : immediate values correspond to all values encoded as integers (such as the constructors of enumerated types, or of course integers themselves), whereas any other value is *boxed* on the heap. The distinction between these two categories is achieved by reserving the least significant bit of each value to indicate its nature : if the least significant bit of an OCaml value is 1, then it is an integer. If it is 0, then the value corresponds to a pointer, since all addresses are even : in a 32-bit representation (resp. 64-bit), all OCaml values have a size of 4 bytes (resp. 8 bytes), meaning that all pointer addresses are multiples of 4 (resp. 8).

This representation allows the components of the virtual machine to quickly distinguish immediate values from allocated values.

Figure 2.3 illustrates the binary representation of OCaml values in the ZAM, on a machine with a 32-bit architecture.

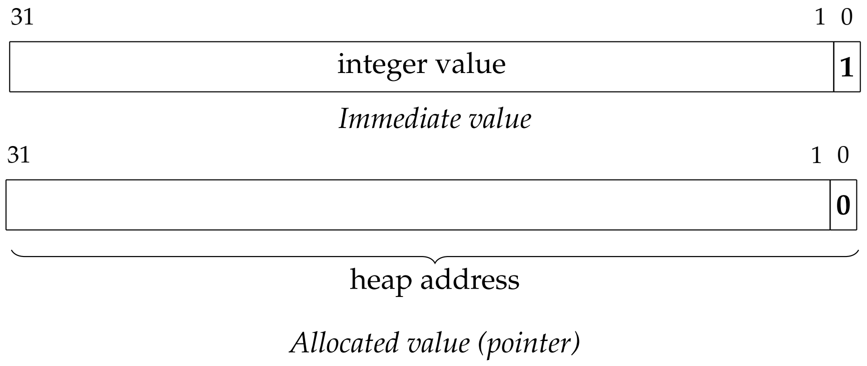


FIGURE 2.3 – Representation of OCaml values in the ZAM on 32-bit architectures

Integer values can then range from -2^{30} to $2^{30} - 1$, while 2^{32} bits are available to represent pointers to the heap. The representation is equivalent on a 64-bit version of the ZAM, in which integers are represented by 63 bits while pointers also end with zero[6].

Header of allocated blocks : OCaml values allocated on the heap—such as closures, floating-point numbers, tuples, records, or values of recursive types (lists, trees, etc.)—are encapsulated in blocks whose first word corresponds to a header. This header contains several pieces of information for the virtual machine interpreter as well as for the garbage collector. In a 32-bit configuration, the header is made up of 22 bits representing the size of the allocated block (excluding the header), 2 color bits required for the proper functioning of the garbage collector, and 8 *tag* bits that indicate the nature of the value (closure, exception, object, etc.). In particular, this tag allows the garbage collector to determine whether or not it should traverse the values contained inside the block :

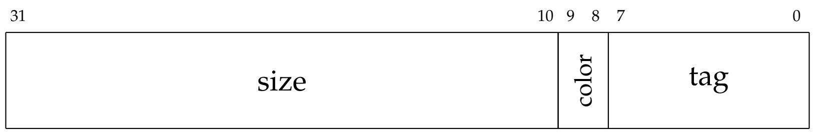


FIGURE 2.4 – Header of an allocated block

The structure of the contents of a block depends on the nature of the value it represents : for example, in the case of a closure (with tag 247), the first value is a pointer to a bytecode segment corresponding to the closure's code, and the subsequent values represent the closure's environment.

6. More precisely, such pointers end with `000` since all addresses are multiples of 8, just as 32-bit addresses end with `00` because they are multiples of 4.

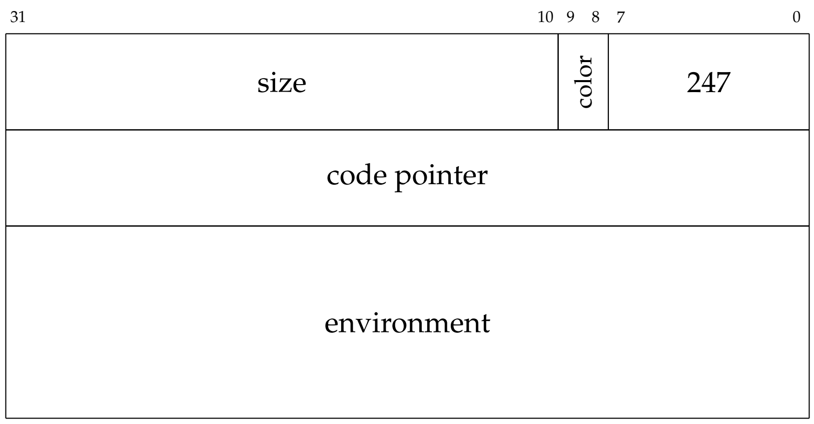
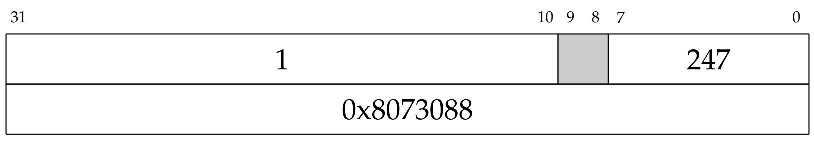
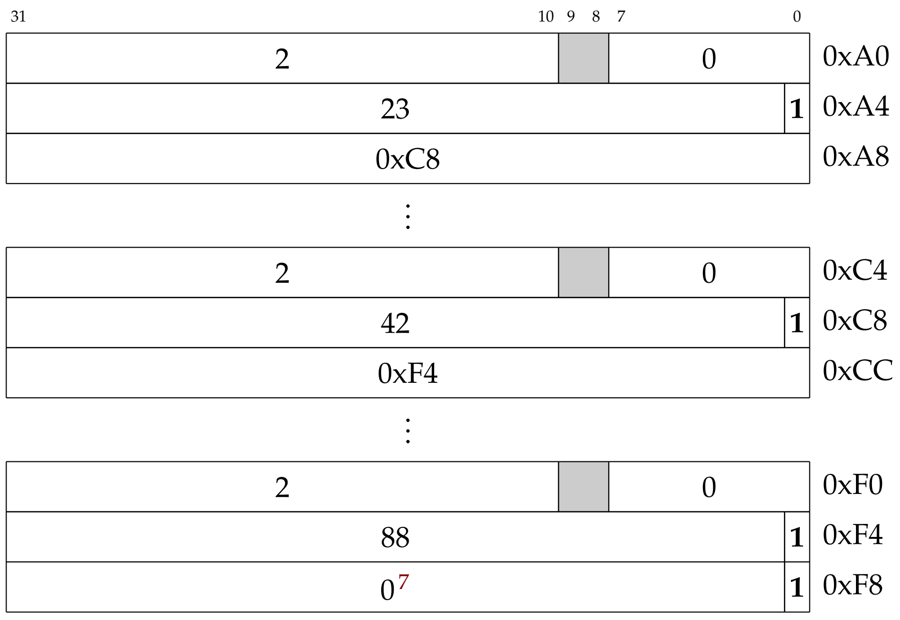


Figure 2.5 – Representation of a closure

For example, the closure generated by the `CLOSUREREC` instruction in the factorial example of Figure 2.1, which corresponds to the `facto` function, is reduced to a block containing a code pointer to the address `0x8073088`, and an empty environment :

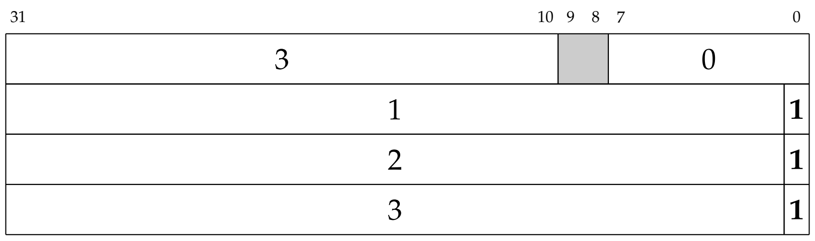


Every data structure is allocated in a block that contains each element of that structure. For example, an element of a list is represented by the value of the element followed by a pointer to the next element (not necessarily contiguous) of the list. It should be noted that a pointer to a block actually points to the first value contained in this block, and not to its header. The list `[23;42;88]` is therefore represented as follows :



Fixed-size data structures, such as arrays or tuples, are represented by blocks containing values stored contiguously in memory. For example, the triple (1,2,3) is represented by the following block :

7. The value 0 here represents the constant constructor `[]` for the empty list.



This representation is also identical for the array `[| 1 ; 2 ; 3 |]` or for a record containing three fields with values 1, 2, and 3. Some type-related information is therefore impossible to recover once a program has been compiled to bytecode (which itself is untyped), and likewise during its execution. Static typing, however, ensures that, for example, an array cannot be mistakenly used as a tuple at runtime.

2.2.5 Runtime library

The ZAM provides a rich standard library, which supports the manipulation of many predefined types, such as lists, mutable arrays, references, and strings. Most modules of this library are defined directly in the OCaml language, with the exception of certain functions that cannot be expressed in OCaml itself and are therefore written in C. An example is the generic function `compare`, which compares two OCaml values of arbitrary type, whether a base type (int, float, etc.) or even a recursive structure, predefined or not (list, tree, etc.).

In addition, memory regions holding values that are no longer used during execution are eventually reclaimed by a garbage collector (GC). The ZAM implements a hybrid incremental and generational GC that relies on two heaps : a fixed-size *minor heap*, into which value blocks are initially allocated, and a *major heap*, which contains blocks that have "survived" (i.e., are still reachable by the program) after a minor heap collection. The minor heap collection uses a *Stop and Copy* algorithm, which copies all live values from the minor heap into the major heap, whereas the major heap collection is based on an incremental *Mark and Sweep* method that traverses the heap in segments, marking all live blocks before reclaiming the others. OCaml's GC also includes a regularly invoked *Mark and Compact* phase, which compacts the major heap to prevent fragmentation.

2.3 Compilation and Execution of an OCaml Program with OMicroB

In this section, we present the operation of the OMicroB virtual machine in light of a detailed description of the different stages required to execute a program on a microcontroller. This description highlights the essential differences between the ZAM and this generic virtual machine, which is specifically designed for programming devices with limited resources.

We therefore describe below the successive transformations applied by OMicroB to an OCaml source program in order to produce an executable that can be stored in a microcontroller's memory. Figure 2.6 provides a schematic representation of this compilation chain : the OCaml source file is first compiled into a bytecode file (a), cleaned (b), then embedded into a C file (c), which is finally compiled together with its interpreter and runtime library (d) into an executable file for the target hardware (e).

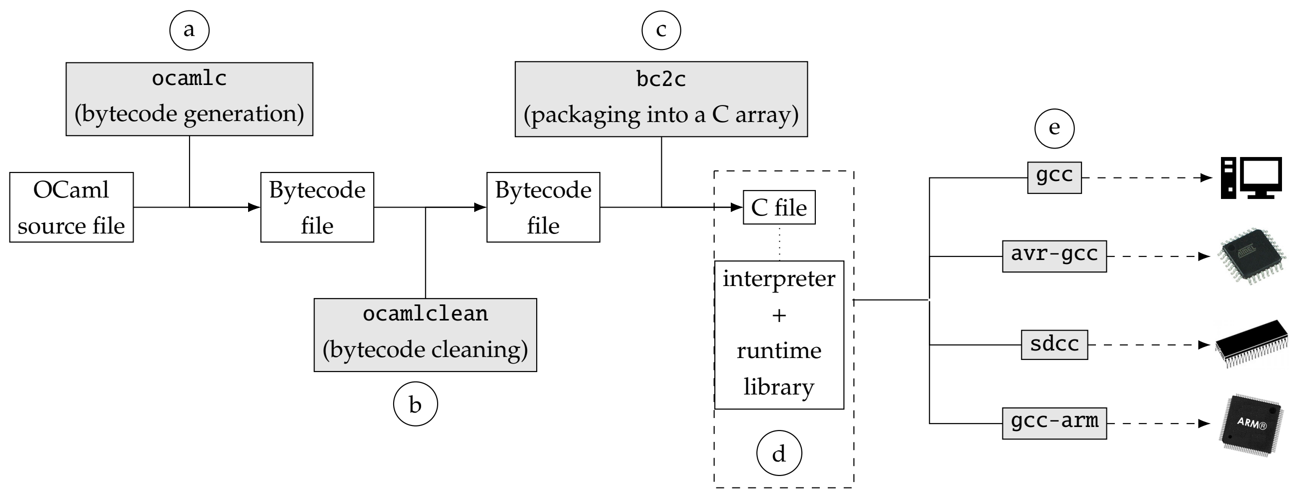


Figure 2.6 – Compilation of an OCaml source file into a microcontroller executable

2.3.1 Representation of an OCaml Program in a C File

The initial steps in producing a microcontroller-executable program from an OCaml source file involve compiling the source into an intermediate representation — OCaml bytecode — which can be interpreted by the virtual machine. In OMicroB, this bytecode is generated, cleaned, and then embedded into a C file containing a representation of the instructions, as well as the declaration of C arrays representing, among other things, the virtual machine's stack and heap.

Generation and Cleaning of Program Bytecode

OCaml programs intended for the OMicroB virtual machine are written using the standard syntax of the language and are therefore fully compatible with the standard OCaml bytecode compiler, `ocamlc`. Such a program is first compiled into a standard bytecode file using this compiler. Leveraging the standard compiler enables the reuse of standard tools from the OCaml ecosystem, such as *ocamldebug*, which can be used to debug OCaml programs by analyzing their bytecode.

The bytecode file generated by `ocamlc` may contain code that is unused by the final program, in particular the initialization code of closures not referenced in the program, induced by the opening of OCaml libraries in the program's source code. For example, a program making use of the `List` module in its source code will result in the generated bytecode including the initialization of all the closures defined in that module. Consequently, the memory footprint of a program may be significantly increased by such initializations, and programs that are *a priori* lightweight may become incompatible with the strict memory constraints inherent to microcontroller usage. In order to reduce the memory consumption of OCaml programs, we make use of the *ocamlclean*[⚓9] tool, originating from the OCaPIC project that preceded our work. This tool is capable of detecting unused blocks in a program through static bytecode analysis, and of removing the initialization code of the unused closures associated with them[8]. The program resulting from this cleaning step is therefore smaller, which makes it suitable for resource-constrained environments.

The memory savings provided by the use of *ocamlclean* can vary greatly from one OCaml program to another, depending on the number of external modules a program depends on, and on its structure. As

8. Note that *ocamlclean* "breaks" dynamic loading of programs since code not used by the main program is deleted. However, this limitation is not problematic in our context of microcontroller programming, as we do not make use of dynamic loading.

an example, consider the OCaml program at the top of Figure 2.7. After compiling this program into a bytecode file using the `ocamlc` bytecode compiler (for version 4.06.0 of the OCaml language), the use of the `-verbose` option of *ocamlclean* produces the output shown beneath the program source code in Figure 2.7. The result is that cleaning this program, which only uses the `map` and `iter` functions from the `List` module, reduces the file size by a factor of 14.8 (from 32.34 kilobytes down to 2.18 kilobytes).

It should also be noted that since the `Pervasives` module (which defines certain basic operations and primitives) is implicitly loaded by every program, even a program that appears almost empty contains a significant number of unused bytecode instructions. The *ocamlclean* tool therefore almost always provides a reduction in program size, even for programs that are a priori very lightweight. For example, on a personal computer, the OCaml program reduced to the single value « `()` » (*unit*) is compiled by *ocamlc* into bytecode containing 2279 instructions, and the use of *ocamlclean* reduces this number to just 81 instructions in total, dividing the program size by 12 (from 18.5 KB down to 1.53 KB).

```
let () =
  let l = List.map (fun x -> x + 1) [1;2;3;4;5] in
  List.iter print_int l
```

```
Statistics:
* Instruction number:     5454  ->    188   (/29.01)
* CODE segment length:   23844  ->    964   (/24.73)
* Global data number:       66  ->     25   (/2.64)
* DATA segment length:     769  ->    376   (/2.05)
* Primitive number:        352  ->      9   (/39.11)
* PRIM segment length:    7065  ->    191   (/36.99)
* File length:           32340  ->   2177   (/14.86)
```

FIGURE 2.7 – Dead code elimination statistics of a program with *ocamlclean*

Generation of a C file

The next stage in the *OMicroB* compilation chain consists of transforming the OCaml bytecode file generated by *ocamlc* into a C file. This file, produced by a tool called *bc2c*, contains a representation of the OCaml program's bytecode as well as the different memory areas required for its execution. Other structures defined in the generated file represent certain data sets used by the OCaml program, such as the table of the program's global variables or a table of C primitives used by the OCaml program, mainly to carry out low-level interactions with its environment.

Global structure of the C program : The C file generated by *bc2c* defines several program-specific elements that are manipulated by the virtual machine interpreter during execution. These elements correspond to the different sections (CODE, PRIM, DATA) of the original bytecode file, as well as the memory used by the OCaml program, represented as C arrays. Figure 2.8 provides a schematic representation of the functioning of *bc2c*. The various elements present in the generated C file are as follows :

— The OCaml program's bytecode, represented as an array of constants.

— The heap of the virtual machine, which contains dynamically allocated values, represented as an array of OCaml values.

— The stack of the virtual machine, also represented as an array of values.
— The table of global variables used in the program.
— The set of primitive functions used by the program. This is represented as an array of pointers to C functions taken either from the standard library or from user-defined external functions.
— The sizes of the generated arrays, together with a set of constant values corresponding to the bytecode instruction numbers, represented as macros at the beginning of the file. In particular, the size of the heap and the size of the stack are chosen by the developer at compile time.

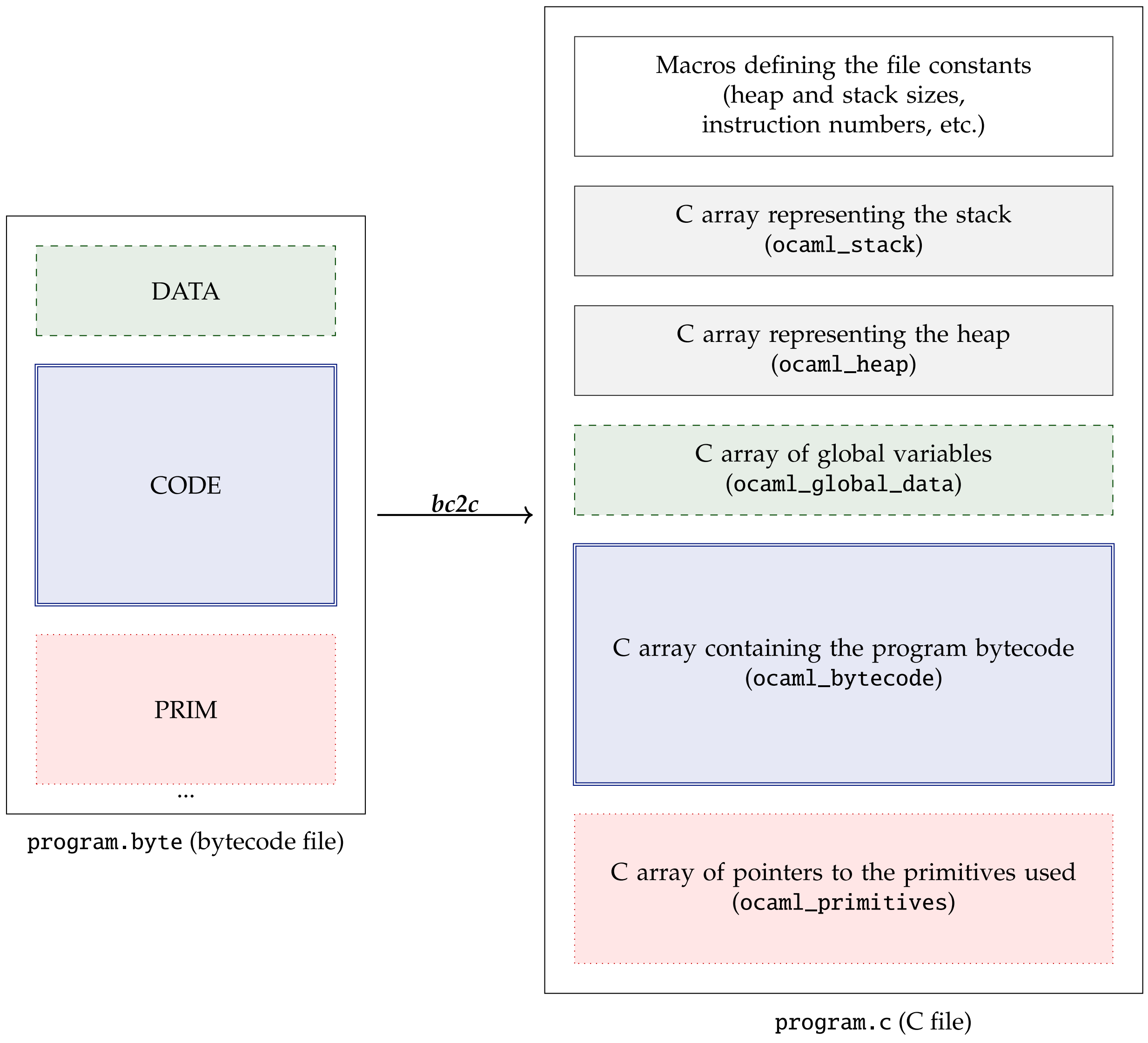

DATA
CODE
PRIM
...
program.byte (bytecode file)
bc2c
Macros defining the file constants (heap and stack sizes, instruction numbers, etc.)
C array representing the stack (ocaml_stack)
C array representing the heap (ocaml_heap)
C array of global variables (ocaml_global_data)
C array containing the program bytecode (ocaml_bytecode)
C array of pointers to the primitives used (ocaml_primitives)
program.c (C file)

FIGURE 2.8 – bc2c : embedding OCaml bytecode in a C file

Bytecode representation : The standard bytecode generated by the OCaml compiler contains only 148 distinct instructions. The arguments of the various bytecode instructions are also statistically rather small (for example, the argument p of the conditional branch instruction `BRANCHIF` p, which represents a relative position in the bytecode, often corresponds to a location situated nearby in the program's bytecode). Aligning bytecode instructions on 32-bit values, as in the standard OCaml virtual machine, would in our use case result in fairly excessive and largely unnecessary resource consumption : memory would be wasted representing empty values consisting primarily of zeros, included solely to achieve 4-byte alignment. Reading bytecode instructions would also be slower, given that the microcontrollers we target generally feature 8-bit processor architectures, which require multiple cycles to read and manipulate larger values.

The tool *bc2c* therefore represents OCaml bytecode as a constant byte array, in which each element corresponds either to an instruction code (or *opcode*) or to an instruction argument. Any argument that does not fit within 8 bits is represented across several consecutive cells of the generated array. This representation of bytecode using 8-bit values improves program execution speed and, more importantly, reduces program size : based on measurements performed on a collection of standard programs, we estimate that bytecode represented as a byte array is approximately 3.5 times smaller than the original bytecode.

To enable the interpreter to distinguish between instructions whose arguments fit into a single byte and those requiring multiple consecutive bytes, *specialized* versions of the relevant bytecode instructions are generated. For example, the instruction `PUSHCONSTINT` k, which pushes the value of the accumulator onto the stack and adds to it the constant integer k, is specialized in OMicroB into three versions capable of handling integers represented on one, two, or four bytes. This specialization of instructions is indicated by appending the annotations `_1B`, `_2B`, and `_4B` to the conventional names of bytecode instructions (for example : `PUSHCONSTINT_1B`, `PUSHCONSTINT_2B`, and `PUSHCONSTINT_4B`).

Figure 2.9 shows the file generated by *bc2c* from the file obtained after compiling the program that computes the factorial of 4, presented in Figure 2.1. To save the microcontroller's RAM, the `PROGMEM` annotation instructs the C compiler[9] to allocate the array representing the bytecode in flash memory, since it is only read during program execution and never modified. The type `opcode_t` is an alias for `int8_t`. The macro `OCAML_BYTECODE_BSIZE` is computed by *bc2c* and represents the total size (in bytes) of the program's bytecode. It should be noted that the addresses of the various bytecode instructions are updated to take into account the offsets introduced by aligning the bytecode to 8-bit boundaries.

2.3.2 Bytecode Interpreter

At runtime, the interpreter of the OMicroB virtual machine traverses the code stored in the array representing the program's bytecode, and manipulates the data stored in the arrays that represent the stack and the heap. This interpreter, whose semantics are identical to those of the ZAM, is responsible for reading the program's bytecode instructions and modifying the program's memory accordingly. The OMicroB interpreter contains the same seven registers as the ZAM (`acc`, `pc`, `trapSp`, `extra_args`, `env`, and `globaldata`), with the notable difference that the `pc` register points to the microcontroller's flash memory, since the program bytecode is stored there in order to reduce RAM consumption. The dynamic elements (stack, heap, and register values) are stored in RAM. Figure 2.10 provides a graphical representation of the interaction between the various components of the OMicroB virtual machine.

Each of the 148 different OCaml bytecode instructions (along with their specialized versions) is handled by the OMicroB interpreter, written in C, which manipulates the various registers and the program's memory. As with the ZAM, the interpreter executes the OCaml program by progressively updating the contents of the virtual machine's registers according to the instructions encountered during

9. This annotation is used by the *avr-gcc* compiler.

```
#define OCAML_STACK_WOSIZE       42
#define OCAML_HEAP_WOSIZE        200
/* (...) */
#define OCAML_BYTECODE_BSIZE    28
#define OCAML_PRIMITIVE_NUMBER   0

value ocaml_stack[OCAML_STACK_WOSIZE];
value ocaml_heap[OCAML_HEAP_WOSIZE];

value ocaml_global_data[OCAML_RAM_GLOBDATA_NUMBER] = { /* ... */ };

PROGMEM opcode_t const ocaml_bytecode[OCAML_BYTECODE_BSIZE] = {
  /*  0 */  OCAML_BRANCH_1B, 17,
  /*  2 */  OCAML_ACC0,
  /*  3 */  OCAML_BRANCHIFNOT_1B, 11,
  /*  5 */  OCAML_ACC0,
  /*  6 */  OCAML_OFFSETINT_1B, (opcode_t) -1,
  /*  8 */  OCAML_PUSHOFFSETCLOSURE0,
  /*  9 */  OCAML_APPLY1,
  /* 10 */  OCAML_PUSHACC1,
  /* 11 */  OCAML_MULINT,
  /* 12 */  OCAML_RETURN, 1,
  /* 14 */  OCAML_CONST1,
  /* 15 */  OCAML_RETURN, 1,
  /* 17 */  OCAML_CLOSUREREC_1B, 1, 0, (opcode_t) -15,
  /* 21 */  OCAML_CONSTINT_1B, 4,
  /* 23 */  OCAML_PUSHACC1,
  /* 24 */  OCAML_APPLY1,
  /* 25 */  OCAML_POP, 1,
  /* 27 */  OCAML_STOP
};

PROGMEM void * const ocaml_primitives[OCAML_PRIMITIVE_NUMBER] = {};
```

FIGURE 2.9 – Example of a file generated by bc2c

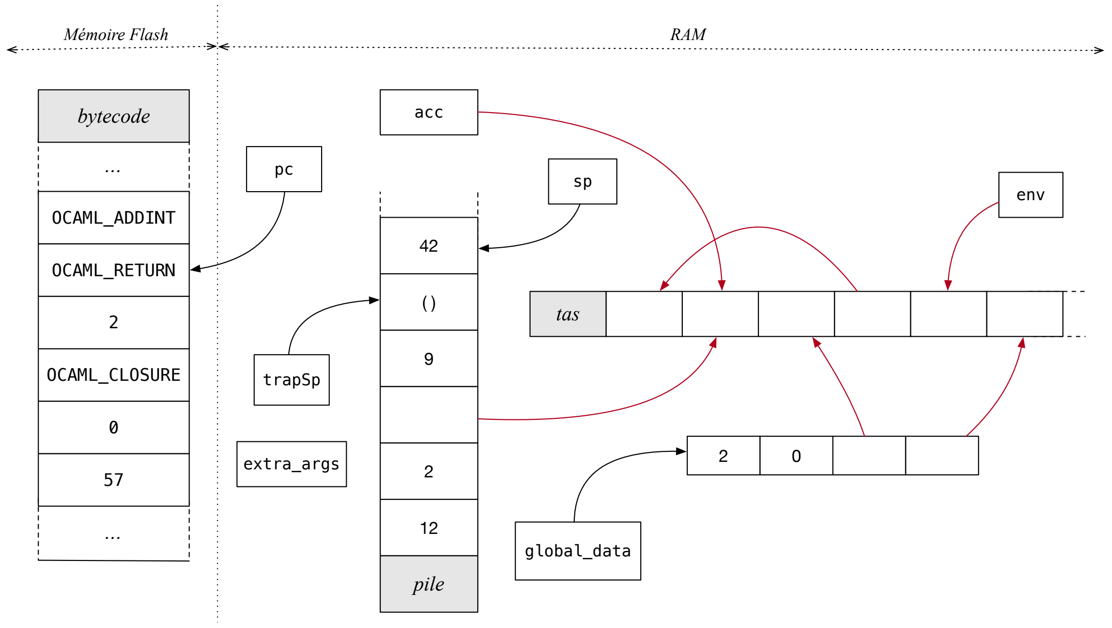


FIGURE 2.10 – The OMicroB virtual machine

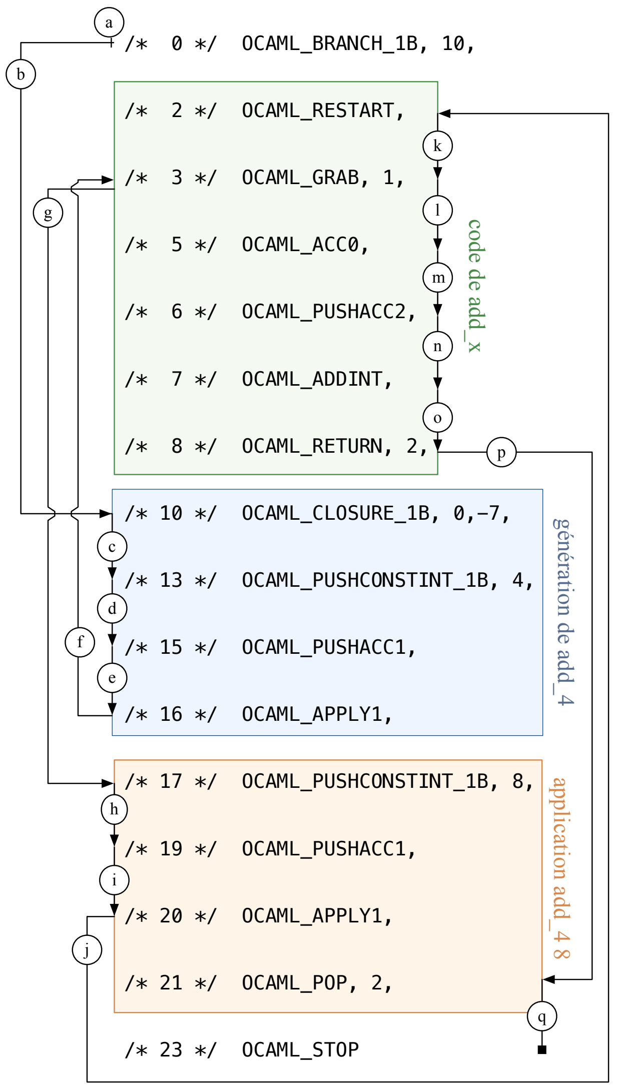
evaluation. We illustrate the interpretation of the bytecode of an example program, shown in Figure 2.11, which generates the functional value $\lambda y.y + 4$ and then applies it to the value 8. Figure 2.12 shows the bytecode generated from the program in Figure 2.11, together with an informal description of the semantics of each instruction, and the detailed evolution of the register values during the interpretation of this bytecode.

```
let add_x x =                 (* add_x receives an integer x and   *)
  (fun y -> y + x)            (* creates a functional value        *)
in                            (* which takes y and computes y + x *)
  let add_4 = add_x 4 in      (* add_4 = (fun y -> y + 4)          *)
  add_4 8                     (* = 12                              *)
```

FIGURE 2.11 – An OCaml program that manipulates a functional value

In this figure, the control flow is represented by arrows annotated with letters, corresponding to the different states of the virtual machine throughout the execution of the program. For each bytecode instruction, we show the state of the registers `pc`, `acc`, `extra_args`, and `env`, as well as the contents of the stack (`sp` points to the first element of the stack) at the end of its interpretation. The registers `trapSp` and `global_data` are not modified during the execution of this program and are therefore not represented. Closures, allocated on the heap, are represented in braces by a code pointer (preceded by the symbol @), followed possibly by a sequence of elements that constitutes their environment. For example, the closure corresponding to `add_4`, whose code begins at instruction 2 and whose environment itself contains a closure (pointing to instruction 3) as well as the integer value 4, is represented as : `{@2 ; {@3} ; 4}`.

Limitations of the 16-bit version : The bytecode instructions related to the handling of polymorphic variants and objects manipulate these variants and object methods in the form of hash codes of their



```
/*  0 */  OCAML_BRANCH_1B, 10,
/*  2 */  OCAML_RESTART,
/*  3 */  OCAML_GRAB, 1,
/*  5 */  OCAML_ACC0,
/*  6 */  OCAML_PUSHACC2,
/*  7 */  OCAML_ADDINT,
/*  8 */  OCAML_RETURN, 2,
/* 10 */  OCAML_CLOSURE_1B, 0,-7,
/* 13 */  OCAML_PUSHCONSTINT_1B, 4,
/* 15 */  OCAML_PUSHACC1,
/* 16 */  OCAML_APPLY1,
/* 17 */  OCAML_PUSHCONSTINT_1B, 8,
/* 19 */  OCAML_PUSHACC1,
/* 20 */  OCAML_APPLY1,
/* 21 */  OCAML_POP, 2,
/* 23 */  OCAML_STOP
```

State	Description
a	Beginning of the program
b	Jump to address (0 + 10)
c	Create a closure (whose code starts at address 10 − 7) in `acc`
d	Push the content of the accumulator (`acc`) and put the constant 4 into `acc`
e	Push the content of `acc` and put `sp[1]` into `acc`
f	Save the current context and jump to the code of the closure contained in `acc`
g	Since `extra_args` < 1, create a closure (whose code starts at address 2) in `acc` and return to the caller
h	Push the content of `acc` and put the constant 8 into `acc`
i	Push `acc` then put `sp[1]` into `acc`
j	Save the current context and jump to the code of the closure contained in `acc`
k	Push the values present in the environment (`env`) and set `extra_args` to the size of `env`
l	Since `extra_args = 1`, decrement `extra_args`
m	Put the value of `sp[0]` into `acc`
n	Push the content of the accumulator and put the value of `sp[2]` into `acc`
o	Pop `sp[0]`, add it to `acc`, and put the result in `acc`
p	End of function : pop two elements and return to the caller
q	Pop two elements
	End of the program

state	pc	acc	stack	env	extra_args
a	0	()	[]	[]	0
b	10	()	[]	[]	0
c	13	{@3}	[]	[]	0
d	15	4	[{@3}]	[]	0
e	16	{@3}	[4; {@3}]	[]	0
f	3	{@3}	[4; @17; []; 0; {@3}]	[{@3}]	0
g	17	{@2; {@3}; 4}	[{@3}]	[]	0
h	19	8	[{@2; {@3}; 4}; {@2}]	[]	0
i	20	{@2; {@3}; 4}	[8; {@2; {@3}; 4}; {@3}]	[]	0
j	2	{@2; {@3}; 4}	[8; @21; []; 0; {@2; {@3}; 4}; {@3}]	[{@2; {@3}; 4}]	0
k	3	{@2; {@3}; 4}	[4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}]	[{@2}]	1
l	5	{@2; {@3}; 4}	[4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}]	[{@2}]	0
m	6	4	[4; 8; @14; []; 0; {@2; {@3}; 4}; {@3}]	[{@2}]	0
n	7	8	[4; 4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}]	[{@2}]	0
o	8	12	[4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}]	[{@2}]	0
p	21	12	[{@2; {@3}; 4}; {@3}]	[]	0
q	23	12	[]	[]	0

FIGURE 2.12 – Evolution of the virtual machine registers during the execution of the program from Figure 2.11

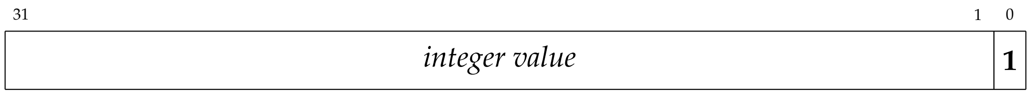
names. These hashes are represented on 31 bits by the standard OCaml bytecode compiler[10], and are thus incompatible with a representation of values on only 16 bits. To avoid making the use of polymorphic variants or objects impossible in the 16-bit version of OMicroB, the VM is provided with a compiler plug-in (implemented as a PPX syntax extension [⚓3]) that automatically translates the names of polymorphic variants and methods into new names whose hash values fit on 15 bits. Indeed, for the names generated by this extension, the compiler's hash function produces values whose 16 most significant bits are all zero. The generation of these names depends only on the original name, which makes it possible to preserve a mechanism of separate compilation of different OCaml modules.

Representation of values

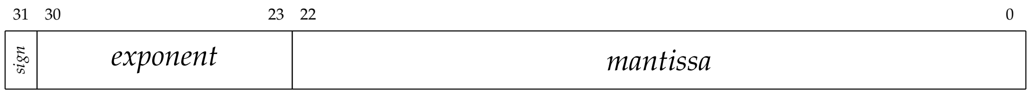
As in the standard OCaml virtual machine, the representation of manipulated values is uniform. The size of these values (16, 32, or 64 bits) is configurable at compilation time, depending on the developer's choice. We consider that the native mechanism of the ZAM, which consists of systematically allocating any non-integer value, and in particular floating-point values, is not necessarily suitable for microcontroller programming. It can indeed be useful, in order to avoid triggering the memory management system during execution, to allocate at program startup the memory space required to represent the variables manipulated by the program, and thus dispense with any dynamic allocation during its execution. We will return more precisely to the advantages of avoiding the triggering of the garbage collector during program execution when we address the calculation of a program's WCET in Chapter 5. In OMicroB, immediate values (non-allocated) can thus be of two types : values represented as integers, or floating-point numbers. The memory representation of values in OMicroB is somewhat different from that of the ZAM. In the following section, we detail the representation of OMicroB values for a 32-bit configuration of the virtual machine.

10. Regardless of the processor architecture – 32 or 64 bits – of the PC that compiles the program.

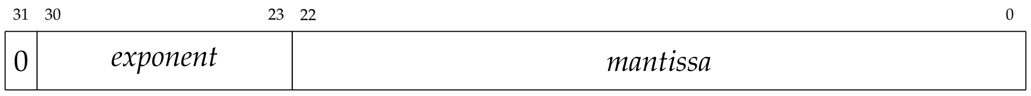
Integer values : Integers are represented in OMicroB in the same way as in the original ZAM : they are encoded on 31 bits, and the least significant bit of each corresponding OCaml value is set to **1** :



Floating-point values : The representation of floating-point values in OMicroB is based on the IEEE 754 standard [STD85]. This standard separates a floating-point number (32 or 64 bits) into three elements : its sign s, its exponent e, and its mantissa m. On 32 bits, a floating-point value is thus represented by one sign bit, 8 bits for its exponent, and 23 bits for its mantissa :

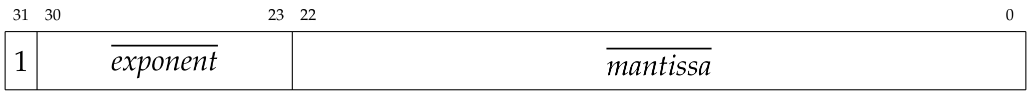


In OMicroB, a *positive* floating-point number is naturally represented in this format on 32 bits :



It should be noted that the representation of an integer and that of a floating-point number can potentially collide. Indeed, if a floating-point number has its least significant bit set to 1, it becomes indistinguishable from an integer. This situation, however, does not cause any issues in our context, thanks to the safety provided by the strict static typing of OCaml programs : at no point can a floating-point number be confused with an integer, and vice versa. All operations in the program are performed on values of compatible types ; for example, there is no risk of adding a value representing an integer to a value representing a floating-point number.

Nevertheless, the standard library provides a polymorphic comparison function, named `compare`, capable of comparing two variables of any type. It is therefore important, given that a floating-point number and an integer are indistinguishable at runtime, that there exists a single consistent way to compare two immediate values (either two integers or two floating-point numbers). This constraint, which effectively allows two floating-point numbers to be compared as if they were integers, requires that the representation of *negative* floating-point numbers be modified, so that the ordering relation between the representation of integers and floating-point numbers is preserved. Consequently, in OMicroB, negative floating-point numbers are represented with the bits of their exponent and mantissa inverted :



This modification of the floating-point representation compared to the standard introduces a slight overhead for the execution of floating-point arithmetic operators : before applying these operators, a "xor" operation is performed on OMicroB's negative floating-point numbers to invert the bits of the mantissa and exponent, thus making them compatible with the standard implementations of floating-point operators.

Heap pointers : The distinction between the different categories of OCaml values is truly useful only when separating immediate values (in OMicroB : integers and floats) from allocated values, which are represented by addresses on the heap. Indeed, a non-ambiguous dichotomy is necessary for the virtual machine's garbage collector so that it can distinguish a pointer to a heap value (which it must visit) from an immediate value (which it must ignore).

In OMicroB, due to the representation of floats as immediate values, it is no longer possible to use the least significant bit of a value to deduce its type (a float may indeed end with 1 or 0). To represent heap pointers, OMicroB employs a trick derived from a particularity of the IEEE 754 standard. In this standard, values for which all exponent bits are set to 1 (i.e., where the exponent equals 128 in a 32-bit representation) correspond to *special* values, which are further divided into two categories :

1. If the mantissa is equal to 0, then this representation corresponds, depending on the sign bit, to the value $\pm\infty$.
2. If the mantissa is nonzero, these values correspond to *NaN* (*Not a Number*) : values used to represent the result of certain invalid operations (for example, division 0/0 or computing the square root of a negative number). These values are very numerous, since any floating-point number with a nonzero mantissa and an exponent equal to 128 is a *NaN* : there are thus $2^{23} - 1$ different *NaN* values in a 32-bit representation. In OMicroB, we exploit this large space of « unused » floating-point values to represent pointers to OCaml values allocated on the heap.

Thus, we represent pointers to the heap by a *NaN* value whose bit pattern begins with **0111 1111 11** :

31	23	22	21 0
0	1 1 1 1 1 1 1 1	1	*heap address*

On a 32-bit representation of the virtual machine, this *NaN-boxing* mechanism [Gud93] with low-bit tagging thus provides us with a 2^{21}-bit space for representing addresses, allowing the allocation of 2^{19} OCaml values (since all addresses are multiples of 4, given that an OCaml value occupies 4 bytes).

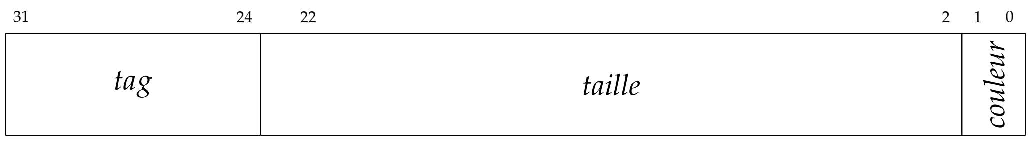
This theoretical 2-megabyte address space is therefore more than sufficient for the hardware we target, which is limited to at most a few tens of kilobytes of RAM, and for which addresses to RAM are often physically limited to 16 bits. For more resource-rich hardware, using a 64-bit configuration of the virtual machine provides a 2^{50}-bit space for representing an address, i.e., over 2^{47} distinct OCaml values, since each then occupies 8 bytes.

Any *NaN* value actually computed by the program is represented in OMicroB by the unique value :

31	30 23	22 0
0	1 1 1 1 1 1 1 1	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

The standard library of OMicroB is adapted so that operations on *NaN*s are consistent with the IEEE 754 standard (which stipulates, for example, that an equality test between two *NaN*s is always false).

Block headers : Finally, the headers of OCaml blocks allocated on the heap are represented on 32 bits, with 8 bits for the *tag*, 22 bits for the size, and 2 bits for the color :



16- and 64-bit representation : The details of value representation in OMicroB for 16-, 32-, and 64-bit configurations are provided in Appendix A. The 64-bit representation is similar to the 32-bit one, extending integer values to 63 bits and representing floating-point numbers according to the IEEE 754 standard in double-precision format. The 16-bit representation is close to the *binary16* representation of the IEEE 754 standard, but allows only 15 effective bits (to distinguish a floating-point number from a pointer) ; we therefore reduce the mantissa of a floating-point number from 10 bits to 9 bits.

2.3.3 Runtime Library

The standard library available with OMicroB includes many common modules from the standard OCaml virtual machine, such as the `List` module, which defines numerous functions for manipulating lists (for example, the `map` function, which applies a function to all elements of a list), the `Queue` module, which allows the representation of mutable first-in-first-out (FIFO) queues, and the `Hashtbl` module, which provides hash table functionality. In addition to these modules common to both implementations, there are modules specific to microcontroller programming, such as the `Avr` module, which defines low-level interaction primitives for *ATmega* microcontrollers, or the `LiquidCrystal` module[11], which allows communication with a liquid crystal display. Conversely, certain modules that are irrelevant or of little use for microcontroller programming, such as the `Unix` module for performing system calls, are not available.

Using OCaml to define the basic primitives for microcontroller configuration and interaction with its environment increases the safety of the programs. Indeed, OCaml's strict typing, together with its support for *Generalized Algebraic Data Types* (GADTs), allows the definition of low-level primitives that enforce certain typing constraints.

For example, the bits of the `SPCR` register (*SPI Control Register*), which configure the serial port of an AVR microcontroller, are represented by the type `spcr_bit` :

```
type spcr_bit = SPR0 | SPR1 | CPHA | CPOL | MSTR | DORD | SPE | SPIE
```

Meanwhile, the bits of the `SPSR` register (*SPI Status Register*), which allows monitoring the state of the serial communication (for example, to check whether a transmission has completed) are represented by the type `spsr_bit` :

```
type spsr_bit = SPI2x | SPSR1 | SPSR2 | SPSR3 | SPSR4 | SPSR5 | SPSR6 | SPIF
```

The microcontroller registers are then represented by a type `'a register` parameterized by the type of bits they contain :

11. Developed by a master's student during an internship focused on constraint-based programming with OMicroB [Pes18].

```
type 'a register =
  | (* ... *)
  | SPCR : spcr_bit register
  | SPSR : spsr_bit register
  | (* ... *)
```

The primitive `set_bit`, of type `'a register -> 'a -> unit`, allows setting a bit of a microcontroller register to 1. At compile-time, any use of this function in a program triggers, thanks to the type system, a check that the bit passed as the second argument indeed belongs to the register passed as the first argument. For example, the incorrect call `set_bit SPCR SPIF` results in an explicit compilation error :

```
Error: This variant expression is expected to have type Avr.spcr_bit
       The constructor SPIF does not belong to type Avr.spcr_bit
Hint: Did you mean SPIE?
```

This mismatch between the name of the bit and the name of the register could not have been detected using simple macros to represent bit names in a low-level language such as C. The verification of the type-correctness of a program illustrates one of the advantages of using a high-level programming language, even for such low-level interactions.

In OMicroB, program memory is automatically managed by a *garbage collector* that implements a standard *stop-and-copy* algorithm. It indeed manipulates two heaps for the values allocated on the heap : a *from-space* and a *to-space*. At each GC execution, the values still used by the program are copied from the from-space to the to-space, after which the roles of the two spaces are swapped (the from-space becomes the to-space, and vice versa).

To achieve this, the memory management algorithm traverses a set of *roots* corresponding to registers and other memory blocks that may contain pointers to heap-allocated values (the arrows pointing to the heap in Figure 2.10 represent such pointers), copies the values to which they point into the new space, and updates each pointer with the new location of the copied values.

The GC takes advantage of the distinction introduced by the value representation in OMicroB : it visits all pointer values encoded via NaN-boxing. Such an algorithm has the advantage of being fast but has the notable drawback of reserving half of the available RAM (the to-space) for its operation. A *Mark and Compact* type garbage collector, which avoids "wasting" half of the heap (at the cost of somewhat slower execution), is also available.

2.3.4 Creation of an Executable

Compilation for the Target Architecture

Compiling an executable program consists of the final step before transferring the program to the target hardware. It makes use of existing C compilers, such as *avr-gcc* or *sdcc*. The C compiler then performs the linking between the C file generated by *bc2c*, the virtual machine interpreter, and its runtime library (standard library and garbage collector). The resulting file is then transferred to the microcontroller (using an appropriate tool such as *avrdude*) and executed on it.

Using the C language as a sort of *portable assembler* allows OMicroB to be deployed with minimal effort on many different architectures. To support a new microcontroller architecture, the virtual machine only requires rewriting the low-level C primitives essential for interacting with the hardware (such as functions to read the microcontroller's flash memory), and does not require any deep modification of the interpreter or other OMicroB components.

Debugging and Simulation of Programs for Microcontrollers

Generating generic C code also allows the virtual machine to run on more conventional hardware. Indeed, the *gcc* compiler can be used on the computer where the program was written to compile it into an executable compatible with that computer's architecture (an x86 or x86-64 executable). This compilation mode allows programs to be simulated on a computer before even transferring them to the target microcontroller, thereby simplifying the debugging process.

In this regard, OMicroB includes tools that simulate the effects of a program with a graphical representation of the microcontroller's input/output pins (Figure 2.13). This simulation allows verifying the consistency of program execution with the expected behavior without forcing the developer to transfer the program to a real microcontroller each time.

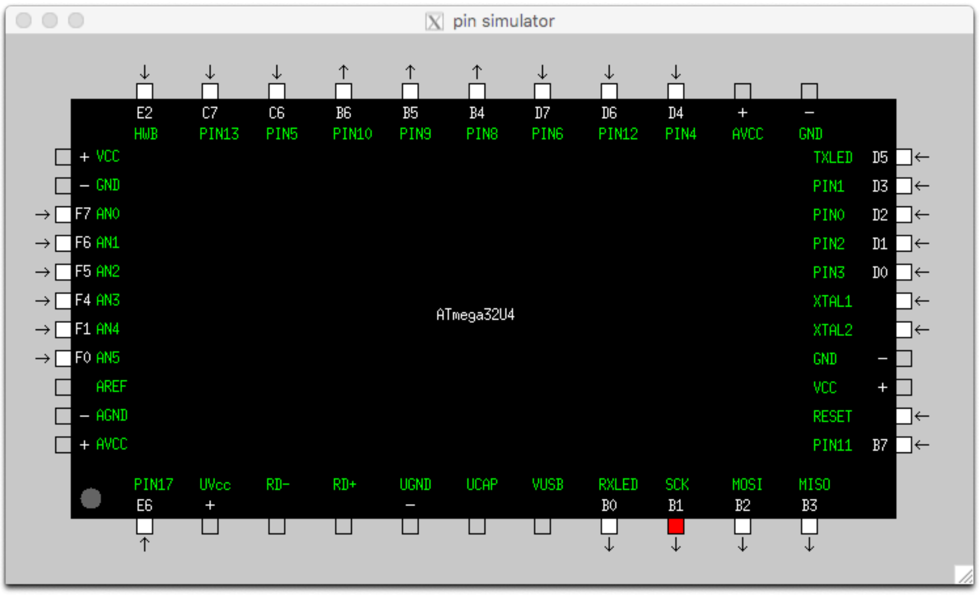
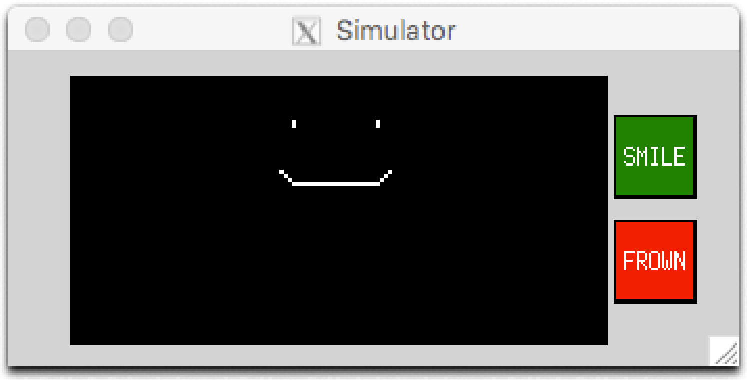


Figure 2.13 – Simulation of the state of the pins of an ATmega32u4 microcontroller

The OMicroB simulator, like the OCaPIC simulator, also allows representing the interactions between the microcontroller and the connected circuit : it indeed provides the programmer with the ability to describe the components attached to the microcontroller (buttons, displays, sensors, . . .) in order to also simulate the effect of the program on them. For example, Figure 2.14 shows the simulation of a small program that displays a smiling face on an OLED screen (*Organic Light-Emitting Diode*) when the user presses the "SMILE" button, and a frowning face when the user presses the "FROWN" button. The various components of the circuit are described in a `circuit.txt` file used by the simulator to produce the appropriate displays.



circuit.txt

```
window width=350 height=150 bgcolor=lightgray title="Simulator"
oled x=30 y=10 column_nb=128 line_nb=8 cs=PIN12 dc=PIN4 rst=PIN6
button x=310 y=100 width=40 height=40 label="SMILE" pin=PIN7 color=green
button x=310 y=50 width=40 height=40 label="FROWN" pin=PIN8 color=red
```

FIGURE 2.14 – Simulation of a small hardware setup

2.4 Optimizations of OMicroB

Considering the significant limitations of the hardware for which the OMicroB virtual machine is intended, we focused our work on reducing the memory footprint of programs executed on microcontrollers. To this end, OMicroB implements several optimizations aimed at limiting the resource consumption of both the virtual machine and the OCaml programs. For example, reducing the size of bytecode instruction *opcodes* from 32 bits to 8 bits constitutes such an optimization. In this section, we present two other optimizations applied to the virtual machine and OCaml programs.

2.4.1 Ahead-of-Time Program Evaluation

The bytecode interpreted at the start of an OCaml program corresponds to a sequence of initializations of values necessary for its execution. First, the program deserializes certain constants (lists, strings, . . .) and allocates them on the heap. Program initialization then continues with the allocation of closures used in the program, the creation of modules, and the computation of global variables. This initialization phase can be slow and consume a non-negligible amount of memory. In particular, the stack depth required to load the modules used by the program can be substantial. For example, the program in Figure 2.15, which defines a class pt, requires at least 106 stack levels to load the modules responsible for object usage in OCaml.

However, the execution of an OCaml program is entirely deterministic during these precomputation phases, regardless of the hardware on which it runs. This determinism persists until the program's first input/output operation, which initiates communication between the program and its environment. It is therefore possible, in order to speed up program startup and especially to reduce memory consumption, to perform the various program initialization steps ahead of its execution on the microcontroller. The interpretation of the bytecode is thus simulated by *bc2c* on the computer on which the program is

```
class pt (x:int) (y:int) =
 object
   method get_x = x
   method get_y = y
 end

let () =
  Avr.delay 1000;
  let p = new pt 1 2 in
  p#get_x
```

FIGURE 2.15 – Example of an OCaml program manipulating objects

compiled up to the first input/output, and the resulting memory state (i.e., the values contained in the heap, the stack, and the program's global variable table) is then directly written into the C code generated by *bc2c*. Consequently, the virtual machine's stack and heap, when producing the C code that embeds the OCaml bytecode, are pre-populated with values corresponding to closures and computed values before the first instruction that begins the interaction between the program and its environment. This process of ahead-of-time evaluation can be seen as a form of partial evaluation [JGS93] applied to the program's bytecode.

In the example of Figure 2.15, enabling ahead-of-time evaluation (which has the effect of pre-executing the program up to the call to the primitive `Avr.delay`[12])) reduces the required stack size for executing the program to only 16 levels.

2.4.2 Tailor-made virtual machine

Most compiled programs do not make use of the entire bytecode instruction set associated with the OCaml language, but only a subset corresponding to the language features actually used in the program. For example, a program that does not use OCaml's object layer will not contain, after compilation to bytecode, the `GETPUBMET` instruction that allows calling a method of an object. Similarly, since many arguments can fit in a single byte, it is rare that instructions specialized for four-byte arguments, such as the `CLOSURE_4B` instruction, are actually used. Therefore, including in the final executable an interpreter capable of handling such instructions absent from the program's bytecode represents a waste of space, especially since microcontroller memory is a scarce resource.

OMicroB avoids this unnecessary memory waste by including, during program compilation, a bytecode interpreter that can only process the instructions actually present in the program's bytecode. To achieve this, directives for the C compiler preprocessor are added to the interpreter code to prevent the compilation of code related to the interpretation of instructions not used in the program. Each opcode present in the program then corresponds to a *macro*, defined in the C code generated by *bc2c*, representing a constant value (chosen so that all values are contiguous). The total size of the executable transferred to the microcontroller is thus reduced.

Figure 2.16 illustrates the structure of the tailor-made interpreter on an excerpt of it : the code responsible for handling the `BRANCHIF_4B` instruction is only compiled if the macro corresponding to this instruction is defined in the C code generated by *bc2c*.

12. This primitive allows pausing the program for the duration indicated by its parameter (in milliseconds).

```
#ifdef BRANCHIF_4B
    case BRANCHIF_4B :
       /* instruction interpretation code */
    break;
#endif
```

FIGURE 2.16 – Tailor-made interpreter

Other optimizations based on the same idea of generality of a C program could also be envisaged. We could indeed go even further in our approach of creating a virtual machine specific to each program, for example by statically replacing each bytecode instruction of a program with the corresponding set of C instructions, thereby eliminating the need for a bytecode interpreter. This approach was followed by the *OCamlCC* tool [MV13], whose first implementation transformed OCaml bytecode into a C program by replacing each bytecode instruction with the associated low-level code through a macro-expansion mechanism. Such an approach provides good execution speed, but the program size grows rapidly because multiple references to a particular bytecode instruction each trigger duplication of the low-level code necessary to execute it. Other approaches, such as CeML [Cha92] or Camlot [Cri92], transform the source code of a program written in an ML language into C code, which can then benefit from C compiler optimizations. The code generated by these solutions is still quite large, and their runtime libraries are generally also sizeable, because they must, for example, include a generic application mechanism to construct closures in the case of partial function application. These approaches are therefore rather reserved for hardware where memory consumption considerations are less important than program execution speed. OMicroB then appears to us as a good compromise between the portability provided by implementing the VM in C and the reduction of program size resulting from representing programs as bytecode, which factors sequences of lower-level instructions. We therefore follow this pragmatic approach, for which execution speeds may be slightly lower, without being particularly penalizing for the applications we target.

Chapter Conclusion

The use of the OMicroB virtual machine constitutes a generic and configurable approach to enable the execution of OCaml programs on a wide variety of devices. In particular, the optimizations implemented in OMicroB allow the interpretation of OCaml bytecode on microcontrollers with very limited hardware resources. This virtual machine is capable of executing OCaml programs on AVR microcontrollers with severely constrained RAM, such as the ATMega325p (2 KB of RAM), ATmega32u4 (2.5 KB of RAM), or ATmega2560 (8 KB of RAM). Several academic projects aiming to port OMicroB to other architectures, such as PIC32 or ARM Cortex-M0 (used by the *micro :bit* development boards, designed by the BBC to support teaching programming to young children [⚓20]), are currently underway, and the initial results are very promising : the first OMicroB ports have indeed been carried out without difficulty [PB19].

The compilation model for OCaml programs considered here, which consists of using a high-level language bytecode together with a generic interpreter, greatly increases program portability : the same OCaml program can thus be easily ported from one device to another. Moreover, this portability allows programs to be easily simulated while taking into account memory constraints imposed by the virtual machine configuration (such as the stack or heap size), providing an accelerated and simplified debugging process.

In order to continue efforts aimed at reducing the RAM footprint of OCaml programs, several techniques are currently under study. In particular, a fine-grained analysis to detect immutable constant values in an OCaml program (such as strings or *sprites* in a video game program) could allow these values to be moved into the program's flash memory at compile time, thereby freeing up the more limited RAM.

Table 2.1 lists the main differences between OMicroB and the standard OCaml virtual machine. In Chapter 7, we will discuss OMicroB's performance in more detail, both in terms of OCaml program execution speed and memory footprint.

Due to its richness and high level of expressiveness, OCaml is a powerful language for describing the algorithmic behavior of programs, and its type safety provides significant guarantees for embedded program development. However, OCaml is currently not well suited for describing concurrent aspects of a program, nor for developing real-time systems. Yet, the embedded programs we consider have many concurrent traits, and the *critical* systems they control very often correspond to real-time systems, for which execution times must be tightly controlled. Consequently, in the next chapter, we introduce the *OCaLustre* language, an extension of OCaml for synchronous programming, which provides a lightweight concurrency model suitable for microcontroller applications. This extension is fully compatible with OMicroB, and thus benefits from both the advantages of OCaml and the optimizations of the virtual machine described in this section.

	ZAM (ocamlrun)	OMicroB
Opcode size	32 bits	8 bits
Number of instructions	148	148 + 46 specialized instr. = 194
OCaml value size	non-configurable (architecture-dependent)	configurable (16, 32, 64 bits)
Address size (address space) in a 32-bit configuration	32 bits (4 GB)	21 bits (2 MB)
Floating-point representation	With encapsulation (heap allocation)	Immediate
GC algorithm	Hybrid (generational / incremental)	Stop and Copy or Mark and Compact

Table 2.1 – Main differences between the standard OCaml virtual machine and OMicroB

3 OCaLustre : Programmation synchrone en OCaml

OCaLustre is a synchronous dataflow extension of the OCaml language, inspired by the Lustre language [CPHP87, Ber86, HCRP91], allowing the use of the synchronous abstraction layer to program concurrency in applications, while leveraging the high-level features of the host language, OCaml, to develop the logical aspects of programs. This extension, with a very light memory footprint, is designed to be executed in conjunction with the OMicroB virtual machine to run lightweight and safe concurrent programs on resource-constrained microcontrollers.

In this chapter, we describe this language extension in detail. In the first section, we present the main traits and principles of the language that enable the creation of synchronous programs in OCaml. In particular, we discuss the *synchronous clock system* implemented in OCaLustre, which allows the presence of certain values to be conditioned during the execution of a program. We then present the specification of the type systems of OCaLustre programs, covering both the "standard" types representing the values carried by program elements, as well as the clock types associated with the notion of *timing* in a synchronous program. Finally, we formally describe the operational semantics of the language, derived from that of the Lustre language.

3.1 Programming in OCaLustre

In this section, we present the main concepts and general features of the OCaLustre language. The purpose of this presentation is to teach a potential OCaLustre developer the different aspects of the language that allow them to write a correct program. Apart from certain specific syntactic variations, these aspects should not surprise readers familiar with the Lustre language (or its derivatives), as the OCaLustre language follows the same semantic foundations.

3.1.1 Language Syntax

In the manner of a Lustre program, an OCaLustre program is composed of *nodes*, which compute streams of values, as well as standard OCaml functions, which handle the algorithmic aspects of programs by leveraging the expressive power of the OCaml language. An OCaLustre node is defined using the `let%node` keyword, followed by its name, the names of its input parameters, and a tuple representing its different output streams (labelled with the keyword `return`). The body of a program is a system of equations solved at each synchronous execution instant of the program, thereby assigning values to the output streams of each node in the program. The equations within the body of a node are stream definitions of the form $y = e$, where y is the name (or a tuple of names) of the defined stream, and e is an expression that evaluates the value of the stream(s) concerned.

We first present a partial version of the language, which we will enrich throughout the discussion. In this version, an expression may correspond to a constant (including the *unit* value), a variable, a constructor of an enumerated type defined in OCaml, the application of the conditional operator (*if-then-else*), the application of an arithmetic or logical operator, or a tuple. Figure 3.1 describes the syntax of this partial version of the language in Backus-Naur Form (BNF) [1].

⟨int⟩	::=	[0 – 9] +
⟨float⟩	::=	[0 – 9] + `.` [0 – 9] ∗
⟨bool⟩	::=	**`true`** | **`false`**
⟨constant⟩	::=	`()` | ⟨int⟩ | ⟨float⟩ | ⟨bool⟩
⟨ident⟩	::=	[*a* – *z*] + [*a* – *zA* – *Z*0 – 9] ∗
⟨lidents⟩	::=	`()` | `(`⟨ident⟩[`,`⟨ident⟩] ∗ `)` | ⟨ident⟩
⟨enum⟩	::=	[*A* – *Z*] + [*a* – *zA* – *Z*0 – 9] ∗
⟨int_op⟩	::=	`+` | `-` | `*` | `/` | **`mod`**
⟨float_op⟩	::=	`+.` | `-.` | `*.` | `/.`
⟨bool_op⟩	::=	`&&` | `||`
⟨comp_op⟩	::=	`<` | `>` | `<=` | `>=` | `=` | `<>`
⟨binop⟩	::=	⟨int_opt⟩ | ⟨float_op⟩ | ⟨bool_op⟩ | ⟨comp_op⟩
⟨unop⟩	::=	`-` | `-.` | **`not`**
⟨eqn⟩	::=	⟨lidents⟩ `=` ⟨expr⟩
⟨leqns⟩	::=	⟨eqn⟩ | ⟨eqn⟩`;`⟨leqns⟩
⟨expr⟩	::=	⟨constant⟩
		| ⟨ident⟩
		| ⟨enum⟩
		| ⟨expr⟩⟨binop⟩⟨expr⟩
		| ⟨unop⟩⟨expr⟩
		| **`if`** ⟨expr⟩ **`then`** ⟨expr⟩ **`else`** ⟨expr⟩
		| ⟨expr⟩`,`⟨expr⟩
		| `(`⟨expr⟩`)`
⟨node⟩	::=	**`let%node`** ⟨ident⟩ ⟨lidents⟩ **`~return:`**⟨lidents⟩ `=` ⟨leqns⟩

FIGURE 3.1 – Partial Syntax of the OCaLustre Language

Here's the English translation of your text :

—

Thus, the example in Figure 3.2 corresponds to the definition of the node named `plus_minus`, which computes a pair corresponding to the sum and difference of its two input parameters, and the associated table represents the evolution of the values of each flow of this node during its execution.

1. The representation of literals and identifiers is simplified here. In the actual implementation of the OCaLustre compiler, any literal or identifier valid in OCaml is also valid in OCaLustre.

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

instant	0	1	2	3	4	5	6	...	i	...
x	3	2	6	32	8	26	67	...	7	...
y	4	12	42	9	22	7	53	...	3	...
p	7	14	48	41	30	33	120	...	10	...
m	-1	-10	-36	23	-14	19	14	...	4	...

FIGURE 3.2 – Example of an OCaLustre node

Initialized Delay

In an imperative language, it is common to assign a value to a variable based on the value it already contains : for example, incrementing a variable (`x := x + 1`) implicitly accesses the variable's « previous » value in order to update its « current » value. OCaLustre, however, is based on a programming model closer to functional programming, in which a variable cannot be modified during program execution. In reality, the dataflow semantics implicitly redefine *new* variables at each synchronous instant, and each of these definitions is itself immutable.

Nevertheless, it is sometimes necessary to access a variable's earlier value in order to assign it a new current value (typically, to increment a counter). In an electronic circuit, it may be useful to access the previously emitted value of a sensor in order to compute its difference with the current valuefor example, to determine the temperature change of a sensor between two instants, with the goal of deducing the rate of growth.

Therefore, in order to allow a stream to access, at instant i, the value of another stream at instant $i-1$, the OCaLustre language provides an *initialized delay* operator, written ⋙ and similar to the `fby` operator (for "*followed-by*") used in the dataflow language Lucid [AW77], as well as in various synchronous languages such as Heptagon [Gér13], Lucid Synchrone [CP99], or Lustre V6 [JRH19].

This operator, used in an expression like `0 ⋙ x`, means that the expression takes the constant on the left of ⋙ (i.e., `0`) as its value at the very first instant of program execution, and then the *previous* value of the expression on the right of the operator (i.e., `x`) for all future instants :

$$k \ggg x \equiv (k, x_0, x_1, x_2, \ldots, x_{i-1}, \ldots)$$

With this operator, it becomes easy to represent sequences of values. For example, the following node computes a stream `n` that corresponds to the sequence of natural numbers :

```
let%node count () ~return:(n) =
   n = (0 ⋙ (n+1))
```

Indeed, the stream `n` has the constant value 0 at the first instant, followed by the previous value of the expression `n+1` at each successive instant.

In the same way, the following example computes the Fibonacci sequence :

```
let%node fibonacci () ~return:f =
  f = (0 >>> ((1 >>> f) + f))
```

The table in Figure 3.3 details the evolution of the different values of the expressions of the `fibonacci` node over time.

instant	0	1	2	3	4	5	6	7	8	...
`f`	0	1	1	2	3	5	8	13	21	...
`(1 >>> f)`	1	0	1	1	2	3	5	8	13	...
`(1 >>> f) + f`	1	1	2	3	5	8	13	21	34	...

FIGURE 3.3 – Computation of the Fibonacci sequence `f` using the initialized delay operator

Let us illustrate the use of this operator in the context of developing programs for microcontrollers. We return to the example program from Section 1.1.2, in which the microcontroller makes an LED blink at regular intervals.

A very simple OCaLustre program can reproduce this behavior. We define the following OCaml enumerated type and OCaLustre node :

```
type light_state = ON | OFF

let%node blinker () ~return:led =
   led = (ON >>> if led = ON then OFF else ON)
```

The flow `led` is therefore initialized with the value « `ON` » to switch the LED on, and at each subsequent synchronous instant the lamp takes the opposite state of its previous one.

The >>> operator also makes it easy to represent flows whose sequence of values repeats cyclically. For example, if we define an enumerated type `tictactoc` as follows :

```
type tictactoc = Tic | Tac | Toc
```

The flow `x` then corresponds to the sequence of values (`Tic`, `Tac` , `Toc` , `Tic` , `Tac` , `Toc` ,..., `Tic` , `Tac` , `Toc` ,...) :

```
x = (Tic >>> (Tac >>> (Toc >>> x)))
```

The node `blinker` can therefore also be written as follows :

```
let%node blinker () ~return:led =
   led = (ON >>> (OFF >>> led))
```

It is also possible to redefine classical operators from the Lustre language using the initialized delay operator :

— The Lustre initialization operator -> can be rewritten using the ⋙ operator as follows :

x -> y ≡ if (true ⋙ false) then x else y

— The delay operator pre can also be defined using ⋙, provided an initial value k (of the correct type) is supplied for the stream in question :

pre x ≡ k ⋙ x

Our choice to use the ⋙ operator in OCaLustre, instead of a combination of these operators, frees the compilation of an OCaLustre program from certain considerations regarding the initialization of streams. Indeed, since for any x the value of the stream pre x at instant 0 is undefined (*nil*), it is usually important to check, at compile time, that the pre operator only appears, in the definition of a stream, to the right of a ->. If this were not the case, the value of the stream at instant 0 would be unknown, and the program would have an indeterminate behavior. This issue of stream initialization, which therefore does not arise in OCaLustre, can nevertheless be resolved statically through the use of a type system capable of representing the position of pre within expressions [CP04].

Locally scoped streams

It should be noted that, in the definition of an OCaLustre node, the equations do not necessarily have to correspond to input or output streams. For readability, or to represent an internal register, or to factorize certain computations, it may be useful to define streams that are considered to have a local scope within the node. In OCaLustre, a local stream is defined without any particular keyword ; its mere absence from the node's signature is enough to limit its scope.

The fibonacci node from the previous section can therefore be rewritten, for improved readability, as a fibonacci2 node defining the local streams sum_last and previous_f :

```
let%node fibonacci2 () ~return:(f) =
  f = (0 ⋙ sum_last);
  previous_f = (1 ⋙ f);
  sum_last = previous_f + f
```

Node application

To organize the different behaviors of a program into distinct blocks of code, we introduce into the language a mechanism of *node application*, similar to what exists in the Lustre language. The call (or application) of a node in an OCaLustre program makes it possible to factorize redundant pieces of code, thereby reducing the memory footprint of a program. It follows the standard OCaml function application syntax : for example, the following node calls the plus_minus node defined earlier, with the pair x,y as input parameters, and with streams a and b as outputs :

```
let%node call_pm (x,y) ~return:(a,b) =
  (a,b) = plus_minus (x,y)
```

It should be noted that, in the language semantics, each call to a node actually corresponds to a call to an *instance* of that node. Each call point to a node implicitly triggers the initialization of an instance of that node and, as a result, no stream is shared between these different call points.

For example, the following node defines *two* counters executing concurrently :

```
let%node two_cpt () ~return:(c1,c2) =
  c1 = count ();
  c2 = count ()
```

The internal counters in each of the calls to `count` are distinct, and the streams `c1` and `c2` increment at the same rate, since each has its own internal register updated at every synchronous instant.

External calls to OCaml functions

The operators built into OCaLustre are by nature very simple : they are limited to basic arithmetic and logical operators, complemented by a few operators referring to time. Yet the host language of this extension, OCaml, has considerable expressive power, through its support for multiple programming paradigms (imperative, functional, object-oriented) and powerful constructions (functors, generalized algebraic data types, polymorphism, . . .), which provide the developer with high-level tools for writing complex programs.

To combine the synchronous aspects of a programwhich govern the interaction between the different software components of an applicationwith its purely algorithmic aspects, which exploit the richness of the host language, OCaLustre is enriched with a **`call`** operator that allows invoking an OCaml function from within an OCaLustre node.

For example, in the following program, the synchronous node `sqrt_cpt` calls the OCaml function `f`, which computes, *within a synchronous instant*, the square root of the stream `a`.

```
let f x = if x > 0.0 then sqrt x else 0.0

let%node sqrt_cpt () ~return:b =
  a = (0.0 ⋙ (a +. 1.0));
  b = call f a
```

It should be noted that the use of **`call`** can be « dangerous » because it opens OCaLustre to all the constructs of the OCaml language, some of which may perform operations incompatible with the semantics of the synchronous model[2], or which may never return (due to an infinite loop or the raising of an exception). It is therefore the programmer's responsibility to guard against such erroneous behaviors. For safety, throughout the rest of this thesis, we assume that **`call`** is only used to execute purely functional code that terminates.

The syntax of OCaLustre expressions presented in figure 3.1 is then extended with the initialized delay operator, node application, and the operator for applying an *n*-ary OCaml function :

2. For example, the behavior of an OCaLustre program in which two concurrent uses of `call` modify the same mutable global variable is undefined.

```
⟨expr⟩  ::=  (. . . )
           | ⟨constant⟩ ⋙ ⟨expr⟩
           | ⟨ident⟩ (⟨expr⟩)
           | call ⟨ident⟩ ⟨expr⟩[ ⟨expr⟩]*
```

3.1.2 Synchronous Clocks

Each component of an OCaLustre program executes concurrently. As a result, every equation defining the value of a flow in a node is evaluated at each synchronous instant, and every expression within these equations is also evaluated at that instant. For example, the following equation triggers the evaluation, in the same instant, of both `e1` and `e2` — the value assigned to x is then chosen according to the value of the boolean `b` :

```
x = if b then e1 else e2
```

This semantics differs from that of general-purpose programming languages, in which the evaluation of the *if-then-else* conditional operator is typically lazy. For example, in OCaml, the evaluation of the following expression only executes the *positive* branch when `b` is true, or the *negative* branch when `b` is false, and the message displayed is `"true"` (resp. `"false"`) :

```
if b then print_string "true" else print_string "false"
```

By contrast, in OCaLustre, both branches of a conditional are executed at every instant : for example, if we define the node `count` that computes a counter :

```
let%node count () ~return:c =
  c = (0 ⋙ (c + 1))
```

Then, in the following equation, the incrementation of the counter in `count ()` is performed *at every instant* :

```
x = if b then count () else 0
```

However, to be able to write non-trivial programs, it is necessary to provide a way to conditionally execute certain parts of a program, thereby recovering the notion of *control flow*. From the perspective of synchronous programming, this amounts to *slowing down* the execution of some branches of the program : that is, ensuring that certain expressions are executed only at particular instants, effectively *downsampling* in time the values of some flows in an OCaLustre program.

Downsampling

In OCaLustre, to ensure that an expression `e` only has a value when a certain boolean flow b is true, we use the annotation `[@when` b`]`. For example, the following equation declares a flow `x` whose value is that of the expression `(y+1)` only when the flow `clk` evaluates to *true*, and which has no value otherwise :

```
x = (y+1) [@when clk]
```

We then say that the flow `x` is *clocked* by the clock `clk`. A clock is a boolean flow that represents the *presence condition* of the flows it governs : when the value of a clock is false, all flows it controls are considered absent (they have no value). In that case, it is forbidden to access the values of those flows. For example, the following OCaLustre program fragment is incorrect :

```
let%node bad_clocks () ~return:(x,y,w) =
   w = 2;
   y = 3;
   x = w [@when b] + y
```

Here, the flow `y`, which is not sampled by any clock (we say it is clocked by the *base clock* — i.e. the fastest clock of the node, which is an implicit parameter of every node), is added to the flow `w`, which is clocked by `b`. Computing the value of `x` when `b` is false has no meaning, since there is "nothing" on the left-hand side of the addition[3]. All binary operators in OCaLustre can only be applied to flows that are clocked by the same clock.

Furthermore, flows sampled by a clock that is itself absent (because its own clock is false) are also absent. In the following example, the flow `z` therefore has no value because its clock `ck2` is absent :

```
let%node sampled_clock () ~return:(ck1,ck2,z) =
   ck1 = false;
   ck2 = true [@when ck1];
   z = 42 [@when ck2];
```

In contrast to *positive* subsampling, OCaLustre also provides a *negative* subsampling annotation `[@whennot _ ]`. This makes it possible to restrict the presence of a value to the condition that a boolean is *false*. Thus, in the following excerpt, the flows `a`, `b`, and `c` have a value only if the flow `clk` evaluates to *false*. We then say that they are sampled by the clock *not* `clk` (or $\overline{\texttt{clk}}$) :

```
a = 2 [@whennot clk];
b = 3 [@whennot clk];
c = a + b
```

3. It should be noted that in the language semantics, *absence* of a value is not equivalent to the presence of the constant *nil*, nor of any default value. It is denoted $\perp$.

The only operator in OCaLustre that allows manipulation of flows not sampled by the same clock is the **merge** operator : this operator "merges" flows sampled by complementary clocks. It takes as first parameter a clock, then a flow sampled positively by that clock, and then a flow sampled negatively by the same clock. In the following example, the flow k takes the value of the flow *i* when *d* is true, and the value of the flow *j* when *d* is false :

```
c = (count () < 5);
d = (true ⋙ false) [@when c];
i = 23 [@when d];
j = 45 [@whennot d];
k = merge d i j
```

The runtime values of the different flows defined in this example are given in Figure 3.4. It should also be noted here that the use of the **@when** operator performs a *sampling* of the values, and does not introduce any *delay* in the computation of those values : the expression (**true** ⋙ **false**) is evaluated at *every* instant of execution (even when c is false)[4], but the corresponding value is only associated with d when c is true. We will return to the distinction between sampling and delay when we discuss the case of conditional applications in Section 3.1.2.

instant	0	1	2	3	4	5	6	7	8	9	10	...
c	true	true	true	true	true	false	false	false	false	false	false	...
d	true	false	false	false	false	⊥	⊥	⊥	⊥	⊥	⊥	...
i	23	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	...
j	⊥	45	45	45	45	⊥	⊥	⊥	⊥	⊥	⊥	...
k	23	45	45	45	45	⊥	⊥	⊥	⊥	⊥	⊥	...

FIGURE 3.4 – Evolution of clocked flow values

The **merge** operator can therefore be seen as the traditional *if-then-else* conditional operator with lazy evaluation from general-purpose languages. It is important to specify that the result (the flow k) is present at the same clock rate as the sampling clock of flows i and j (the flow d — whether its value is *true* or *false*), and is not faster : since d is only present when c is true, k is present only under the same condition.

The syntax of OCaLustre expressions, extended with sampling annotations and the **merge** operator, is now complete :

```
⟨expr⟩  ::=  (...)
          |  merge ⟨ident⟩ ⟨expr⟩ ⟨expr⟩
          |  ⟨expr⟩ [@when ⟨ident⟩]
          |  ⟨expr⟩ [@whennot ⟨ident⟩]
```

4. This is due to Lustre's *substitution principle*, which states that if a flow x is equal to *expr*, then any occurrence of *expr* in the program can be replaced by x, and vice versa, without changing its semantics. The expression (**true** ⋙ **false**) could therefore be extracted into a new variable, evaluated at each instant.

Special case : constants

In OCaLustre, constants have the particularity of being clocked by an arbitrary clock : we can consider that their clock type is polymorphic, and therefore they are compatible with any clock. This distinction stems from the desire to make OCaLustre programs simpler to write and to read. For example, in the following :

```
let%node ex_const (a,b) ~return:x =
  c = a;
  d = b [@when c];
  v = 12 [@whennot c];
  x = merge c d (4 [@whennot c] + v)
```

an expression such as `(4 [@whennot c])` is relatively "heavy," and is not useful given the context of its use (as the third parameter of a `merge` whose first parameter is c). Here, the clock of `4` can only be $\bar{c}$. In the same way, since the variable `v` is added to a value clocked by $\bar{c}$, its clock is also $\bar{c}$.

The same node is thus more readable when the constants are stripped of their clock annotations :

```
let%node ex_const (a,b) ~return:x =
  c = a;
  d = b [@when c];
  v = 12;
  x = merge c d (4+v)
```

Conditional application

When a flow is defined as the result of a node call, it is important to distinguish between the subsampling of the call parameters and the subsampling of the return value. Indeed, subsampling an input value serves to *condition the execution* of the node call, and thus to slow down the execution frequency of the called node ; whereas subsampling the return value of a node call merely serves to *limit the presence* of this value.

For example, below we give the definition of a node `counter` which computes the sequence of natural numbers, modulo `n` :

```
let%node counter n ~return:c =
  c = (0 >>> (c + 1)) mod n
```

We then define a node `call_counter` which computes two flows : a flow `a`, corresponding to the call to `counter` with subsampling of its parameters, and a flow `b`, which corresponds to the call to `counter` with subsampling of its return value :

```
let%node call_counter clk ~return:(a,b) =
  a = counter (10 [@when clk]);
  b = (counter 10) [@when clk]
```

From the point of view of their clocks, the flows a and b are indistinguishable, each of these flows being present only when the flow `clk` is true. However, their semantics differ : for flow b, the call `(counter 10)` is executed at every instant (whatever the value of `clk`), and it is the return value of this call that is subsampled by `clk`. In contrast, for flow a, the execution of `(counter 10)` is performed only when `clk` is true. The counter is then incremented only when `clk` is true, and the execution frequency of `counter` is potentially slower than that of `call_counter`. The table in Figure 3.5 illustrates the difference between these two semantics.

instant	0	1	2	3	4	5	6	...
`clk`	true	false	true	true	false	false	true	...
`counter (10 [@when clk])`	0	⊥	1	2	⊥	⊥	3	...
`(counter 10) [@when clk]`	0	⊥	2	3	⊥	⊥	6	...

FIGURE 3.5 – Difference between subsampling the parameters and subsampling the return value.

Subsampling the parameters of a node therefore makes it possible to *slow down* the call frequency of this node, and thus to make its internal state evolve more slowly. Such behavior will be referred to in this thesis as *conditional application*.

A classic example of using conditional application is the watch example [Pla89]. In this example, the different calls to `counter` are conditioned by clocks running at increasingly slower frequencies, allowing one to simulate the computation of hours, minutes, and seconds[5] :

```
let%node watch (sec) ~return:(hour,min,h,m,s) =
  s = counter (60 [@when sec]);
  min = (s = 60);
  m = counter (60 [@when min]);
  hour = (m = 60);
  h = counter (24 [@when hour])
```

Each call to `counter` is executed at a distinct frequency :
— The flow `sec` is incremented each time its clock *sec* is true.
— The flow `min` is incremented each time *min* is true (60 times more slowly than `sec`).
— The flow `hour` is incremented each time *hour* is true (60 times more slowly than `min`).

The notion of *conditional application* must be distinguished from that of *conditional activation*, used for example in the Scade language (where it is implemented by the operator `condact` in versions prior to Scade 6, and then by the construction `activate/every` [Dor08]). Conditional activation corresponds to a conditional application of a node combined with a projection : if the clock of the conditional application is false, then the associated flow retains the value calculated the last time its clock was true (and if it has never yet been true, the flow has a default value explicitly provided by the programmer).

5. The reason for the presence of the clocks h and m in the output tuple of the node will be explained when we discuss clock typing in Section 3.1.3.

This mechanism of conditional activation can be implemented in OCaLustre by combining conditional application, the merge operator, and the delay operator. For example, the following node performs the conditional activation of a call to the node `counter` :

```
let%node condact_counter (c) ~return:(x) =
  x = merge c (counter (10 [@when c])) ((0 >>> x) [@whennot c])
```

Determining the application condition

The condition governing the execution of a node call depends on the clock of its parameters. Thus, the following equation triggers the execution of `counter 60` only when `sec` is true :

```
s = counter (60 [@when sec])
```

However, when a node has several parameters, the question arises as to under which condition a call should be executed. Indeed, this condition is not limited to requiring that all the parameters of the call be present. For example, let us start by defining a node `swap_merge` that merges two flows (`x` and `y`) on complementary clocks (`c` and *not* `c`), using the **`merge`** operator with its parameters in reverse order :

```
let%node swap_merge (x,y,c) ~return:z =
  z = merge c x y
```

And now consider the following equation :

```
m = swap_merge (x [@when c], y [@whennot c], c)
```

The parameters of the call to `swap_merge` can never all be present at the same instant, since `x` and `y` are clocked by complementary clocks (when `x` is present, `y` is absent, and vice versa). Requiring the presence of all parameters to condition the call would therefore be too restrictive. In reality, the activation of a node occurs only when certain parameters are present : namely, those that, in the context of the called node, are on the base clock (the fastest one). Since the base clock of a node is the fastest clock of that node, the presence of the parameters on this clock then serves as the guard for the execution of that node. For example, the single parameter `n` of the node `counter` is on the base clock. It is thus the presence of this parameter that conditions the call to `counter`.

In the node `swap_merge`, it is the last parameter that lies on the base clock of the node. The flow `c` must therefore be present for the call `swap_merge (x,y,c)` to be executed.

This semantics, similar to that of Heptagon, is slightly more flexible than that of Lustre : in Lustre, the first parameter must be on the base clock, and the other parameters may potentially be on slower clocks[6]. When calling a node, the presence or absence of the first parameter is then checked to condition its execution.

6. The node `swap_merge` would therefore, in Lustre, have *c* as its first parameter.

In the following section, we present a typing discipline that models the system of synchronous clocks. This system of synchronous clocks, whose consistency can be verified at compile-time for an OCaLustre program, governs the *well-clockedness* of programs in this language.

3.1.3 Clock Typing of Programs

In an OCaLustre program, each expression has a clock defined either explicitly (through the use of the annotations `@when` and `@whennot`), or implicitly (such as the base clock of the node, or a clock resulting from the composition of two sub-sampled nodes). Just like the data types of OCaLustre expressions, the information concerning the clocks of each flow is entirely deducible at compile time. Consequently, the synchronous clocks of OCaLustre can be represented by a type system, statically assigning to each expression of the language a *clock type*. This system is then associated with a set of rules representing the consistency of a program's *clocking*. Thus, a *well-clocked* program corresponds to a program whose clock types respect the rules of this system, and to say that a program is well-clocked amounts to saying that no access to absent values is possible during a synchronous instant, regardless of the values of the flows manipulated by the program. This section introduces the notion of synchronous clock typing through several examples that highlight its specificity, while the formal description of this type system will be presented in the following section.

A clock type is associated with a grammar, presented in Figure 3.6. This grammar makes it possible to represent the base clock of a node, or clocks corresponding to sub-samplings of values :

ck	$::=$		clock
	$\mid$	$\bullet$	base clock
	$\mid$	$ck\ \mathbf{on}\ x$	positive subsampling
	$\mid$	$ck\ \mathbf{onnot}\ x$	negative subsampling
	$\mid$	$ck \times ck'$	tuple
	$\mid$	$ck \rightarrow ck'$	function

FIGURE 3.6 – Clock types in OCaLustre

— A flow that is not sampled by any clock within a node is considered to be on the node's base clock. Following the formalism adopted for the Heptagon language [Gér13], the base clock type is denoted by the following symbol : $\bullet$.
— A flow positively clocked by a flow x (itself of type ck) has the clock type ck **on** x.
— A flow negatively clocked by a flow x' (itself of type ck') has the clock type ck' **onnot** x'.
— An n-tuple of flows corresponds to an n-tuple of clocks.
— An instance of a node has a *functional* type of the form *input clocks* $\rightarrow$ *output clocks*.

The signature of a node is associated with a *type scheme* (denoted ω), that is, a type in which variables may be globally quantified (as defined by Damas and Milner in [DM82]), in the same way as in the work of Colaço and Pouzet [CP03]. Similar to the classical data typing described in the previous section, the clock type of a node is an arrow type, where the left-hand side corresponds to the clock type of its input flows, and the right-hand side to the clock type of its output flows. The signature is annotated with the names of the inputs $\vec{x}$ and outputs $\vec{y}$ of the corresponding node. The associated type scheme is quantified by the base clock of the node :

$$\omega ::= \forall \bullet . (\vec{x} : ck) \rightarrow (\vec{y} : ck')$$

For example, the node `sampler` sub-samples its first parameter with its second parameter :

```
let%node sampler (x,c) ~return:y =
  y = x [@when c]
```

Since in the body of this node there is no information indicating that the parameters x or c are sub-sampled by a slower clock than the base one, these parameters are considered to be clocked by the node's base clock. The output value, however, is explicitly conditioned on the presence of c.

The signature of sampler is therefore :

$$\forall \bullet \, . \, ((x : \bullet) \times (c : \bullet)) \rightarrow (y : \bullet \ \mathbf{on} \ c).$$

It should be noted that the name c in the output's clock type does not mean that the parameter c is itself a type, but rather that the flow c is a Boolean value that sub-samples the flow y. In what follows, we will call such variable names that appear in the type of a node « *supports* ».

The operator **merge** combines flows whose clock types are complementary : (ck **on** x) and (ck **onnot** x). The resulting value is clocked *by the clock of* x, since the merge operator in some sense allows one to « climb back up » one level of clock. For example, let us consider the following example, which defines a node that applies the merge operator to its input parameters :

```
let%node merger (c,a,b) ~return:d =
  d = merge c a b
```

The flow c is on the node's base clock (●). The flows a and b must be on two complementary clocks, clocked (the first positively and the second negatively) by c. The node merger therefore has the following signature :

$$\forall \bullet \, . \, ((c : \bullet) \times (a : \bullet \ \mathbf{on} \ c) \times (b : \bullet \ \mathbf{onnot} \ c)) \rightarrow (d : \bullet)$$

In OCaLustre, if an output flow is locally sub-sampled by a clock defined in the body of a node, then it is mandatory that its clock is also returned by the corresponding node. For example, the following node calls the node sampler, and returns the flow v sub-sampled with its clock c :

```
let%node sampler2 (u) ~return:(v,c) =
  v = sampler (u,c);
  c = (true ⋙ (false ⋙ c))
```

The signature of this node is therefore $\forall \bullet \, . \, (u : \bullet) \rightarrow ((v : \bullet \ \mathbf{on} \ c) \times (c : \bullet))$. This constraint ensures the consistency of a program's clocking : indeed, the use of a flow clocked by a clock whose own clock type would be unknown (since it is defined locally within a node and therefore inaccessible from the outside) leads to incomplete information regarding the presence conditions of this flow.

For example, let us suppose that the definition of the following node were allowed :

```
let%node sampler_wrong (u) ~return:(v) =
  v = sampler (u,c);
  c = (true ⋙ (false ⋙ c))
```

The signature of `sampler_wrong` would therefore be $\forall \bullet\,.\,(u : \bullet) \rightarrow (v : \bullet \textbf{ on } c)$. But since the value of c is known only locally within this node, the clocking information for v is incomplete from the point of view of any node that calls `sampler_wrong` : this is a case where the local variable c escapes its scope. For example, in the following code fragment, the variable that conditions the presence of `a_sampled`, which is supposed to be passed as the first parameter of the **`merge`** operator in the definition of the flow `b`, is inaccessible since it is local to `sampler_wrong` :

```
let%node call_sampler_wrong (a) ~return:(b) =
  a_sampled = sampler_wrong (a);
  b = merge XX a_sampled 32 (* issue: XX est unknown *)
```

Node Calls and Substitution of the Base Clock

The clock typing of node calls has a semantics similar to the typing of function application in a programming language that implements a polymorphic type system : the base clock ($\bullet$) then plays the role of a *type variable*, and is therefore instantiated for each call.

In the classical type system of such a programming language (for example that of OCaml), if a function `f` has the type scheme $\forall\alpha.\alpha \rightarrow \alpha$, then in the application `f 2` the type variable α is instantiated with the type `int`. The type of `f` in this context is therefore `int` → `int` (and the result of the application is then of type `int`).

Similarly, if we define the following node in OCaLustre :

```
let%node plus_one x ~return:y =
  y = (x + 1)
```

Then, the signature of this node is $\forall \bullet\,.\,(x : \bullet) \rightarrow (y : \bullet)$, and the base clock ($\bullet$) can be substituted by a slower clock during its instantiation. For example, let us consider the following equation :

```
y = plus_one (42 [@when c])
```

Since the expression `42 [@when c]` has the clock type ($\bullet$ **on** c), then, after instantiation of the base clock, the clock type of `plus_one` in this context will be the following :

$$(\bullet \rightarrow \bullet)\,[\bullet := \bullet \textbf{ on } c] = (\bullet \textbf{ on } c) \rightarrow (\bullet \textbf{ on } c)$$

The flow `y` will therefore have the clock type of the result of the call : ($\bullet$ **on** c).

Node Calls and Substitution of Variable Names

A particular feature of the type system for synchronous clocks is that certain clock types contain variable names (for example the variable c in the type $(\bullet\ on\ c)$). These names, which we will call *supports*, correspond to the *formal* names of the flows used as clocks in the equations.

Of course, the supports appearing in node signatures may differ from the *actual* names of the arguments in a call. For example, let us consider the signature of the node `sampler` :

$$\forall \bullet\ .\ ((x : \bullet) \times (c : \bullet)) \rightarrow (y : \bullet\ \mathbf{on}\ c)$$

The output flow of the node `sampler` has the clock type $(\bullet\ \mathbf{on}\ c)$, with c being the support of its clock (corresponding to the second parameter of `sampler`). However, in the following example the name of the second argument of the call to `sampler` is not `c`, but `d` :

```
let%node call_sampler (d) ~return:w =
  w = sampler(8,d)
```

It would therefore be incorrect to assign to `w` the clock type $(\bullet\ \mathbf{on}\ c)$, since the flow `c` is undefined in the body of this node. This difference between the *formal* parameter names of a node and the *actual* names of its arguments makes it necessary to substitute the supports with the actual names of the corresponding variables.

Thus, in the previous example, the type of this instance of `sampler` becomes :

$$((\bullet \times \bullet) \rightarrow (\bullet\ \mathbf{on}\ c))[c := d] = (\bullet \times \bullet) \rightarrow (\bullet\ \mathbf{on}\ d)$$

and the clock type of `w` is therefore $(\bullet\ \mathbf{on}\ d)$. Consequently, the signature of the node `call_sampler` is :

$$\forall \bullet\ .\ (d : \bullet) \rightarrow (w : \bullet\ \mathbf{on}\ d)$$

Finally, let us consider the following example, which combines the substitution of the base clock and the substitution of supports :

```
let%node call_sampler_slower (c,e) ~return:w =
  d = c [@when e];
  w = sampler(8 [@when e],d)
```

The parameters of the call to `sampler` are each sub-sampled by `e`, so the base clock of this call has the type $(\bullet\ \mathbf{on}\ e)$. Moreover, the name of the second argument of this call is `d`; consequently, the type assigned to `sampler` in the equation `w` is then :

$$((\bullet \times \bullet) \rightarrow (\bullet\ \mathbf{on}\ c))\,[c := d]\,[\bullet := \bullet\ \mathbf{on}\ e] = ((\bullet\ \mathbf{on}\ e) \times (\bullet\ \mathbf{on}\ e)) \rightarrow ((\bullet\ \mathbf{on}\ e)\ \mathbf{on}\ d)$$

The node `call_sampler_slower` therefore has the following signature :

$$\forall \bullet\ .\ (d : \bullet) \times (e : \bullet) \rightarrow (w : (\bullet\ \mathbf{on}\ e)\ \mathbf{on}\ d)$$

3.2 Specification of the OCaLustre Language Formalized with Ott and Coq

This section is devoted to the description of a formal specification for the OCaLustre language. In particular, we detail rules relating to the semantics and typing of an OCaLustre program. These various rules are expressed using an intermediate representation of an OCaLustre program, called the *normal form*. Moreover, the structure of an OCaLustre program is organized as a header containing OCaml definitions (of types or functions), followed by a list of node definitions. The OCaml header (which is unrestricted) will not be formalized in this manuscript, and the normal form will only concern synchronous nodes and the equations they contain.

The grammar adopted, as well as each of the inference rules described in this section, have been formalized with the Ott tool [SNO+10]. This tool is intended for the definition of grammars and evaluation rules for programming languages, and is capable of extracting from this definition files readable by several languages and proof assistants. Thus, the formal specification of the type systems and semantics described in this section is based on the Coq proof assistant [Tea19]. The display of the various inductive rules in this section is generated from the extraction of Ott into LaTeX.

3.2.1 Representation of a Synchronous Program in Normal Form

The various operations and analyses that we will formally describe in the remainder of this manuscript rely on a structured representation of programs. This representation, the *normal form*, results from a procedure carried out by the OCaLustre compiler, which applies several static transformations intended to homogenize the structure of programs in order to simplify their subsequent analyses and transformations. In particular, the normal form makes it possible to extract sub-expressions that induce the use of registers during the compilation of a program. We will describe in detail the process of converting a program into normal form (or *normalization*) when we present the different stages of OCaLustre program compilation in Chapter 4.

Grammar

For the sake of consistency with related work, the grammar associated with the normal form representation of OCaLustre programs is inspired by the formalism introduced in [BDPR17] and the definition of the CoreDF language syntax[7]. An OCaLustre program is therefore first converted from an abstract syntax tree (AST), resulting from the parsing of the source program, into a *normalized* AST, whose components are derived from the following grammar :

— A synchronous program in OCaLustre corresponds to a list of node definitions :

$program, \vec{nodes}$	::=		programme
	|	$\varnothing$	programme vide
	|	$node;;\vec{nodes}$	liste de nœuds

7. Unlike CoreDF, our representation supports the definition of nodes with multiple output values.

— A node is defined by its name f, a list of variable names $\vec{x}$[8] representing the input flow(s), a list of names $\vec{y}$ corresponding to the output flows, and a (non-empty) list of equations $\vec{eqn}$ that constitutes its body :

$node$	::=		définition de nœud
	|	$\textbf{node} f (\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn}$	
$\vec{x}$, $\vec{y}$	::=		noms de variables
	|	()	liste vide
	|	y	unique variable
	|	$y, \vec{y}$	multiples variables
$\vec{eqn}$	::=		liste d'équations
	|	$[eqn]$	unique
	|	$eqn;\ \vec{eqn}$	multiples

— An equation, of the form *name(s) = expression*, corresponds to the definition of one or more flow variables. These flows may correspond to a control expression *ce*, the use of the operator $\ggg$ (for consistency with other synchronous languages, this operator will be represented as **fby** in normal form), the application of a node f with as parameter a (non-empty) list of expressions $\vec{e}$, or the use of the `call` operator to apply an OCaml function :

eqn	::=		équation
	|	$y = ce$	expression
	|	$y = k\ \textbf{fby}\ e$	fby
	|	$\vec{y} = f(\vec{e})$	application
	|	$y = \ \textbf{call f}\ e_0\ e_1 \ldots e_{n-1}$	application d'une fonction OCaml
$\vec{e}$	::=		liste d'expressions
	|	$[e]$	unique
	|	$e, \vec{e}$	multiples

— Control expressions correspond to the use of the conditional operator **if** − **then** − **else**, the merge operator **merge**, or « simple » expressions *e* :

ce	::=		expression de contrôle
	|	e	expression
	|	$\textbf{merge}\ x\ ce\ ce'$	fusion
	|	$\textbf{if}\ e\ \textbf{then}\ ce\ \textbf{else}\ ce'$	alternative

8. Arrows placed above a given syntactic category will, throughout this manuscript, denote a list of elements of that category.

— Finally, the « simple » expressions correspond to the value *unit* (represented by the standard notation « () »), constants (k), variables (x), constructors of enumerated types (X_i), application of the positive or negative sub-sampling operators (**when** and **whennot**), application of a unary prefix arithmetic operator ($\square$), as well as the application of a binary infix arithmetic or logical operator ($\diamond$) :

e	::=		expression
	|	$()$	unit
	|	k	constante
	|	x	variable
	|	X_i	constructeur de type
	|	$e \diamond e'$	opération binaire
	|	$\square\, e$	opération unaire
	|	e **when** x	échantillonnage positif
	|	e **whennot** x	échantillonnage négatif
k	::=		constante
	|	*int_literal*	entier
	|	*float_literal*	flottant
	|	*bool_literal*	booléen
$\diamond$	::=		opérateur binaire
	|	$\diamond_{int}$	opérateur entier
	|	$\diamond_{float}$	opérateur flottant
	|	$\diamond_{comp}$	opérateur de comparaison
	|	$\diamond_{bool}$	opérateur booléen
$\square$	::=		opérateur unaire
	|	$-$	opposée entière
	|	$-.$	opposée flottante
	|	**not**	négation booléenne

3.2.2 Typing and Clocking

The type system of a programming language is a set of rules that assign a *type* to the various constructs of a program in that language. The types most commonly considered in computer programming concern those that represent the nature of the data being manipulated : integers, booleans, floating-point numbers, lists of integers, functions, object instances, and so on.

However, a type system can also define rules relating to other aspects of a program beyond the nature of the computed values. For example, it is possible to design a type system that is not dedicated to representing the nature of the values « carried » by different variables and expressions, but rather to the properties emphasized in certain kinds of applications — such as the security level of different program constructs [Wal00].

The set of typing rules, which govern the consistency of a type system, makes it possible to distinguish well-typed programs (which respect these rules) from programs whose typing is incorrect (for example, those in which forbidden implicit type conversions are performed).

In this section, we present the typing of an OCaLustre program through the lens of two type systems that provide very different kinds of information. The first type system we describe is a classical one, which assigns to values or expressions of the language data types representing the nature of the information being transmitted : flows of integers, flows of booleans, flows of floating-point numbers, and so on. This type system relies on the underlying host language's type system and therefore shares all its characteristics : thus, in OCaLustre, data are, just as in OCaml, strongly typed, and their types are statically inferred at program compilation.

The second type system we consider in this section is the type system of synchronous clocks. This system, unlike the previous type system, does not represent the nature of the values manipulated by the various components of a program, but rather the presence (or absence) of values. The typing of synchronous clocks, like the typing of data in OCaLustre, is a strong static type system, with inference of flow clocks at program compilation.

The advantage of implementing such systems, with this strong static typing constraint, is that a very large number of potential bugsstemming from inconsistencies in the programmer's writing of certain expressions of the languagecan be detected at compile time, even before the program has been deployed to the microcontroller. These type systems, each of which governs a distinct aspect of the consistency of OCaLustre programs, thus contribute to the safety of OCaLustre programs.

3.2.3 Typing Rules of Programs

Like its target language OCaml, the OCaLustre language extension is a statically typed programming language, implementing a mechanism of type inference at compilation. Each variable of an OCaLustre program has a type whose nature can be determined at compile time. It should be noted, however, that the type system of OCaLustre is monomorphic, unlike the polymorphic type system of OCaml.

Saying that a flow $x \equiv (x_1, x_2, ..., x_i, ...)$ is of type « *flow of t* » (simply written « t ») means that all the elements of the sequence of values it corresponds to are of type t. Just like homogeneous lists in the OCaml language, a flow cannot « carry » values whose type changes over time. Consequently, creating a flow whose value could change type from one instant to the next, such as the flow $(0, 2, 4, \mathit{true}, 10, 4.5, 14, \dots)$, is therefore impossible in OCaLustre.

The types manipulated in OCaLustre can be base types (int, bool, float), arrow types (the types of nodes), enumerated types (represented by the set of their constructors X_i), or n-tuples of types (corresponding to the type of a list of expressions) :

$$t \ ::= \ \mathit{unit} \mid \mathit{int} \mid \mathit{bool} \mid \mathit{float} \mid t \rightarrow t' \mid \{X_1, \dots, X_n\} \mid t_1 \times ... \times t_n$$

From the normal form representation of OCaLustre programs, we then formally define the conditions that govern the well-typing of a program.

Let Γ be a local typing environment, that is, a function which associates to each variable of an OCaLustre node its type : for example, if the flow x is of type int, then $\Gamma(x) = \mathit{int}$. We write $\Gamma \vdash e : t$ for the predicate (called a *typing judgment*) indicating that, in the typing environment Γ, the expression e has type t.

We then define from such judgments inference rules representing the consistency of typing expressions in OCaLustre. Such rules, of the form

$$\frac{P_1 \qquad \dots \qquad P_n}{Q}$$

make it possible to establish that, from the *premise* conditions P_1 to P_n, one can deduce the *judgment* Q.

Typing of expressions : Figure 3.7 presents the typing rules for OCaLustre expressions. For example, the typing rule for the application of an integer arithmetic operator $\diamond_{int}$ (for instance, the operator +) states that such an operation can only be performed between two integers, and the result is itself an integer :

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \diamond_{int} e_2 : int}$$

It should be noted here that, for the typing rule of comparison operators, it is assumed that no comparison of values of functional type is carried out in an OCaLustre program. This is left to the responsibility of the programmer, who must ensure not to perform such an operation[9]. Since we rely on the execution of the OCaml code generated by compiling OCaLustre, the comparison function is polymorphic, and its use on values of functional types raises an exception. Any exception in an OCaLustre program results in the termination of its execution.

Moreover, a judgment of the form $t = \{X_1, \dots, X_n\}$ means that the type t has previously been defined (in the OCaml header) as an enumerated type composed of the constructors X_1 to X_n.

Finally, we distinguish the typing rules of a « simple » expression from those of a conditional expression by annotating the symbol « *turnstile* » ($\vdash$) in the case of rules concerning conditional expressions as follows : $\vdash_{ce}$.

The rules concerning the typing of OCaLustre equations and nodes are given in Figure 3.8. In the following, we review them in order to describe them in detail.

Typing of equations : Let $\mathcal{G}$ be a *global* typing environment that associates to each OCaml function and each OCaLustre node its type (an « arrow » type of the form *input type* → *output type*). An equation in the body of an OCaLustre node corresponding to the application of another node $\vec{y} = f(\vec{e})$ is well-typed if and only if f has the signature of a type $t_1 \rightarrow t_2$, the parameter $\vec{e}$ of the application is of type t_1, and the result $\vec{y}$ is of type t_2 :

$$\frac{\Gamma \vdash \vec{y} : t_2 \qquad \mathcal{G}(f) = t_1 \rightarrow t_2 \qquad \Gamma \vdash \vec{e} : t_1}{\mathcal{G}, \Gamma \vdash \vec{y} = f(\vec{e})}$$

9. Just as the application of partial functions via `call`, or division by zero, are prohibited.

$\boxed{\Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash () : \mathit{unit}} \qquad \frac{}{\Gamma \vdash \mathit{int\_literal} : \mathit{int}} \qquad \frac{}{\Gamma \vdash \mathit{bool\_literal} : \mathit{bool}} \qquad \frac{}{\Gamma \vdash \mathit{float\_literal} : \mathit{float}}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \frac{t = \{X_1, ..., X_n\}}{\Gamma \vdash X_i : t}$$

$$\frac{\Gamma \vdash e_1 : \mathit{int} \quad \Gamma \vdash e_2 : \mathit{int}}{\Gamma \vdash e_1 \diamond_{\mathit{int}} e_2 : \mathit{int}} \qquad \frac{\Gamma \vdash e_1 : \mathit{bool} \quad \Gamma \vdash e_2 : \mathit{bool}}{\Gamma \vdash e_1 \diamond_{\mathit{bool}} e_2 : \mathit{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathit{float} \quad \Gamma \vdash e_2 : \mathit{float}}{\Gamma \vdash e_1 \diamond_{\mathit{float}} e_2 : \mathit{float}} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \diamond_{\mathit{comp}} e_2 : \mathit{bool}}$$

$$\frac{\Gamma \vdash e : \mathit{int}}{\Gamma \vdash -e : \mathit{int}} \qquad \frac{\Gamma \vdash e : \mathit{float}}{\Gamma \vdash -.e : \mathit{float}} \qquad \frac{\Gamma \vdash e : \mathit{bool}}{\Gamma \vdash \mathbf{not}\, e : \mathit{bool}}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma(x) = \mathit{bool}}{\Gamma \vdash e\, \mathbf{when}\, x : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma(x) = \mathit{bool}}{\Gamma \vdash e\, \mathbf{whennot}\, x : t}$$

$\boxed{\Gamma \vdash \vec{e} : t}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash [e] : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash \vec{e} : t'}{\Gamma \vdash e, \vec{e} : t \times t'}$$

$\boxed{\Gamma \vdash_{ce} ce : t}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash_{ce} e : t} \qquad \frac{\Gamma \vdash e_1 : \mathit{bool} \quad \Gamma \vdash_{ce} ce_2 : t \quad \Gamma \vdash_{ce} ce_3 : t}{\Gamma \vdash_{ce} \mathbf{if}\, e_1\, \mathbf{then}\, ce_2\, \mathbf{else}\, ce_3 : t}$$

$$\frac{\Gamma(x) = \mathit{bool} \quad \Gamma \vdash_{ce} ce_1 : t \quad \Gamma \vdash_{ce} ce_2 : t}{\Gamma \vdash_{ce} \mathbf{merge}\, x\, ce_1\, ce_2 : t}$$

FIGURE 3.7 – Typing rules for OCaLustre expressions

$$\boxed{\Gamma \vdash \vec{y} : t}$$

$$\frac{}{\Gamma \vdash () : \mathit{unit}} \qquad \frac{\Gamma(y) = t}{\Gamma \vdash y : t} \qquad \frac{\Gamma \vdash y : t \quad \Gamma \vdash \vec{y} : t'}{\Gamma \vdash y, \vec{y} : t \times t'}$$

$$\boxed{\mathcal{G}, \Gamma \vdash \mathit{eqn}}$$

$$\frac{\Gamma \vdash y : t \quad \Gamma \vdash_{ce} ce : t}{\mathcal{G}, \Gamma \vdash y = ce} \qquad \frac{\Gamma \vdash y : t \quad \Gamma \vdash k : t \quad \Gamma \vdash e : t}{\mathcal{G}, \Gamma \vdash y = k\,\mathbf{fby}\,e}$$

$$\frac{\Gamma \vdash \vec{y} : t_2 \quad \mathcal{G}(f) = t_1 \rightarrow t_2 \quad \Gamma \vdash \vec{e} : t_1}{\mathcal{G}, \Gamma \vdash \vec{y} = f(\vec{e})}$$

$$\frac{\Gamma \vdash y : t_n \quad \mathcal{G}(\mathtt{f}) = t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_{n-1} \rightarrow t_n \quad \Gamma \vdash e_0 : t_0 \quad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_{n-1} : t_{n-1}}{\mathcal{G}, \Gamma \vdash y = \ \mathbf{call}\,\mathtt{f}\, e_0\, e_1 \ldots e_{n-1}}$$

$$\boxed{\mathcal{G}, \Gamma \vdash \vec{eqn}}$$

$$\frac{\mathcal{G}, \Gamma \vdash \mathit{eqn}}{\mathcal{G}, \Gamma \vdash [\mathit{eqn}]} \qquad \frac{\mathcal{G}, \Gamma \vdash \mathit{eqn} \quad \mathcal{G}, \Gamma \vdash \vec{eqn}}{\mathcal{G}, \Gamma \vdash \mathit{eqn};\ \vec{eqn}}$$

$$\boxed{\mathcal{G} \vdash \mathit{node} : t}$$

$$\frac{\mathcal{G}, \Gamma \vdash \vec{eqn} \quad \Gamma \vdash \vec{x} : t_1 \quad \Gamma \vdash \vec{y} : t_2}{\mathcal{G} \vdash \mathbf{node}\, f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn} : t_1 \rightarrow t_2}$$

$$\boxed{\mathcal{G} \vdash \mathit{program}}$$

$$\frac{}{\mathcal{G} \vdash \varnothing} \qquad \frac{\mathcal{G} \vdash \mathbf{node}\, f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn} : t_1 \rightarrow t_2 \quad (f : t_1 \rightarrow t_2) \uplus \mathcal{G} \vdash \vec{nodes}}{\mathcal{G} \vdash \mathbf{node}\, f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn};;\vec{nodes}}$$

FIGURE 3.8 – Typing rules for equations and nodes

An equation corresponding to the application of the « followed-by » operator is well-typed provided that the constant on the left of the operator $\ggg$ and the expression on the right of it are of the same type, and that the variable on the left of the equality operator is also of the same type :

$$\frac{\Gamma \vdash y : t \quad \Gamma \vdash k : t \quad \Gamma \vdash e : t}{\mathcal{G}, \Gamma \vdash y = k\,\mathbf{fby}\,e}$$

The application of an OCaml function is of type t_n if and only if the OCaml function itself is of type $t_0 \rightarrow \cdots \rightarrow t_{n-1} \rightarrow t_n$ and its successive parameters are of type t_0 through t_{n-1} :

$$\frac{\Gamma \vdash y : t_n \quad \mathcal{G}(\mathtt{f}) = t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_{n-1} \rightarrow t_n \quad \Gamma \vdash e_0 : t_0 \quad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_{n-1} : t_{n-1}}{\mathcal{G}, \Gamma \vdash y = \ \mathbf{call}\,\mathtt{f}\, e_0\, e_1 \ldots e_{n-1}}$$

It should be noted, however, that this rule induces a slight semantic shift : we move from *flows* to OCaml functions, which are supposed to be able to manipulate only « standard » values. For example, a value of type « flow of `int` » in OCaLustre is seen by the called OCaml function as a value of type `int`. This distinction poses no problem, however, since at each instant of the program a flow has only a single value (a « flow of `int` » is therefore reduced to a simple `int`)[10]. Formally, the operator **`call`** can be seen as a *map* function that « *lifts* » an OCaml function so that it can be applied to flows :

$$\texttt{call}\ f\ e_0\ e_1\ \ldots\ e_{n-1} \equiv (map\ f)\ e_0\ e_1\ \ldots\ e_{n-1}$$

Any other equation $y = ce$ is well-typed if the expression ce associated with it is well-typed and if the variable y has the same type in Γ :

$$\frac{\Gamma \vdash y : t \qquad \Gamma \vdash_{ce} ce : t}{\mathcal{G}, \Gamma \vdash y = ce}$$

A list of equations (the body of a node) is well-typed as soon as all the equations it contains are well-typed :

$$\frac{\mathcal{G}, \Gamma \vdash eqn}{\mathcal{G}, \Gamma \vdash [eqn]} \qquad \frac{\mathcal{G}, \Gamma \vdash eqn \qquad \mathcal{G}, \Gamma \vdash \vec{eqn}}{\mathcal{G}, \Gamma \vdash eqn;\ \vec{eqn}}$$

Typing of a node and of a synchronous program : The type of an OCaLustre node is similar to the type of a function in OCaml. It is an arrow type $t_1 \to t_2$, with t_1 being the type of the flows that can be passed to it as inputs, and t_2 the type of the values it computes as outputs :

$$\frac{\mathcal{G}, \Gamma \vdash \vec{eqn} \qquad \Gamma \vdash \vec{x} : t_1 \qquad \Gamma \vdash \vec{y} : t_2}{\mathcal{G} \vdash \mathbf{node} f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn} : t_1 \to t_2}$$

Finally, a synchronous program (i.e. a list of nodes) is well-typed provided that each node it contains is well-typed :

$$\frac{}{\mathcal{G} \vdash \varnothing} \qquad \frac{\mathcal{G} \vdash \mathbf{node} f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn} : t_1 \to t_2 \qquad (f : t_1 \to t_2) \uplus \mathcal{G} \vdash \vec{nodes}}{\mathcal{G} \vdash \mathbf{node} f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn};;\vec{nodes}}$$

The typing judgment of a synchronous program is initially invoked with the typing environment corresponding to the OCaml functions previously defined in the header of the program. The typing environments for OCaml functions and for nodes are therefore shared, and we will assume that the namespaces between functions and nodes are distinct, so that there are no name conflicts (which would result in shadowing).

The typing rules of the OCaLustre extension are sufficiently close to the typing rules of OCaml that a well-typed OCaLustre program remains well-typed after its translation into OCaml, and that the generation of an ill-typed OCaml program indicates that the original OCaLustre program is itself ill-typed. Consequently, thanks to the *typing correctness with respect to the translation into OCaml* (the proof of which we will present in Section 5.1), the rules stated in this section do not need to be explicitly checked

10. This phenomenon also appears when « wiring » an OCaLustre program to functions that compute the inputs and outputs of a synchronous instant.

by the OCaLustre compiler in order to maintain program safety. Indeed, we take advantage of the target language, whose compiler already performs static type checking and type inference. In our case, this analysis is therefore carried out after translating OCaLustre code into standard OCaml code. The OCaml compiler is then able to point out typing errors in programs, including those originating from OCaLustre code.

Moreover, since OCaLustre is built with tools that are part of the OCaml program compilation ecosystem, it is fully compatible with other powerful static analysis tools, such as Merlin [BRS18], which provides numerous features to simplify the development of OCaml programs (such as a context-sensitive auto-completion mechanism, or direct access to typing information) by integrating into various text editors (Vim, Emacs, Atom, . . .). Such compatibility allows the OCaLustre user to receive immediate and precise feedback on the location of errors related (among other things) to typing consistency and program scheduling, directly within their preferred text editor.

3.2.4 Rules for Well-Clocked Programs

The consistency of the synchronous clock type system governs the *well-clocking* of an OCaLustre program. This well-clocking is defined by a set of typing rules whose satisfaction ensures the clocking safety of a program.

Just as with the formalization of OCaLustre program typing, we define a *clocking judgment* as a predicate of the form $\mathcal{H}, \mathcal{C} \vdash eqn$, meaning that in a global clocking environment $\mathcal{H}$ (which associates each node name with its clock) and a local clocking environment $\mathcal{C}$ (which associates a clock with each variable name of a node), an equation eqn is well-clocked. Similarly, a judgment $\mathcal{C} \vdash e : ck$ states that, in the typing environment $\mathcal{C}$, the expression e has clock type ck.

From such typing judgments, we then describe inference rules that formally establish the necessary conditions for the well-clocking of programs, nodes, equations, and expressions. These rules are derived from a type system that considers clocks as abstract data types [CP03], implemented in the *Lucid Synchrone* language. Our solution, less expressive because it does not allow the existence of *multiple* clock type variables within a node but only a single one (the base clock •), nevertheless greatly simplifies the model of separate compilation of nodes, without significantly limiting the expressive power of the language. The type system of OCaLustre is therefore closer to the type system of the synchronous language *Heptagon*, halfway between the clock system of Lustre and that of Lucid Synchrone.

The clocking rules for OCaLustre expressions are presented in Figure 3.9, and those concerning equations and nodes are given in Figure 3.10. In the following, we describe the main rules that govern the well-clocking of an OCaLustre program.

In OCaLustre, a constant is always well-clocked ; it is considered well-typed regardless of the clock type :

$$\frac{}{\mathcal{C} \vdash k : ck}$$

The clock type of a variable comes from the typing environment :

$$\frac{\mathcal{C}(x) = ck}{\mathcal{C} \vdash x : ck}$$

$\boxed{\mathcal{C} \vdash e : ck}$

$$\frac{}{\mathcal{C} \vdash () : ck} \quad \frac{}{\mathcal{C} \vdash k : ck} \quad \frac{}{\mathcal{C} \vdash X_i : ck} \quad \frac{\mathcal{C}(x) = ck}{\mathcal{C} \vdash x : ck}$$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash \Box\, e : ck} \quad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e' : ck}{\mathcal{C} \vdash e \diamond e' : ck}$$

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\,\textbf{when}\,x : ck\,\textbf{on}\,x} \quad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\,\textbf{whennot}\,x : ck\,\textbf{onnot}\,x}$$

$\boxed{\mathcal{C} \vdash \vec{e} : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash [e] : ck} \quad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash \vec{e} : ck'}{\mathcal{C} \vdash e, \vec{e} : ck \times ck'}$$

$\boxed{\mathcal{C} \vdash_{ce} ce : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash_{ce} e : ck}$$

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash_{ce} ce : ck \quad \mathcal{C} \vdash_{ce} ce' : ck}{\mathcal{C} \vdash_{ce} \textbf{if}\,e\,\textbf{then}\,ce\,\textbf{else}\,ce' : ck} \quad \frac{\mathcal{C} \vdash x : ck \quad \mathcal{C} \vdash_{ce} ce : ck\,\textbf{on}\,x \quad \mathcal{C} \vdash_{ce} ce' : ck\,\textbf{onnot}\,x}{\mathcal{C} \vdash_{ce} \textbf{merge}\,x\,ce\,ce' : ck}$$

FIGURE 3.9 – Clocking rules for expressions

The use of any operator requires its operands to have the same clock type. For example, the following rule, which defines the well-clocking of an arbitrary arithmetic or logical operator ($\diamond$), states that in the clocking environment $\mathcal{C}$, if each operand of the operator $\diamond$ is clocked by the clock ck, then its result is itself clocked by ck :

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e' : ck}{\mathcal{C} \vdash e \diamond e' : ck}$$

The use of the **merge** operator requires its second and third parameters to be clocked positively and negatively, respectively, by its first parameter. The result is on the clock of the first parameter :

$$\frac{\mathcal{C} \vdash x : ck \quad \mathcal{C} \vdash_{ce} ce : ck\,\textbf{on}\,x \quad \mathcal{C} \vdash_{ce} ce' : ck\,\textbf{onnot}\,x}{\mathcal{C} \vdash_{ce} \textbf{merge}\,x\,ce\,ce' : ck}$$

The application of a node $\vec{y} = f(\vec{e})$ is well-clocked provided that the clock type $ck_1 \rightarrow ck_2$ is an *instance* of the signature ω of f (i.e. a clock type in which the substitutions required for conditional application and for the consistency of support names have been carried out), that the parameters $\vec{e}$ of the application are of clock type ck_1, and that the result has clock type $\vec{ck_2}$:

$$\frac{\mathcal{H}(f) = \omega \quad ck'_1 \rightarrow ck'_2 = inst(\omega) \quad \mathcal{C} \vdash \vec{e} : ck'_1 \quad \mathcal{C} \vdash \vec{y} : ck'_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y} = f(\vec{e})}$$

$\boxed{\mathcal{C} \vdash \vec{y} : ck}$

$$\frac{}{\mathcal{C} \vdash () : \bullet} \qquad \frac{\mathcal{C}(y) = ck}{\mathcal{C} \vdash y : ck} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{C} \vdash y, \vec{y} : ck \times ck'}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash eqn}$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash_{ce} ce : ck}{\mathcal{H}, \mathcal{C} \vdash y = ce} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e_0 : ck \quad \mathcal{C} \vdash e_1 : ck \quad \dots \quad \mathcal{C} \vdash e_{n-1} : ck}{\mathcal{H}, \mathcal{C} \vdash y = \textbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1}}$$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e : ck}{\mathcal{H}, \mathcal{C} \vdash y = k\ \textbf{fby}\ e} \qquad \frac{\mathcal{H}(f) = \omega \quad ck'_1 \to ck'_2 = inst(\omega) \quad \mathcal{C} \vdash \vec{e} : ck'_1 \quad \mathcal{C} \vdash \vec{y} : ck'_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y} = f(\vec{e})}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash \vec{eqn}}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash eqn}{\mathcal{H}, \mathcal{C} \vdash [eqn]} \qquad \frac{\mathcal{H}, \mathcal{C} \vdash eqn \quad \mathcal{H}, \mathcal{C} \vdash \vec{eqn}}{\mathcal{H}, \mathcal{C} \vdash eqn;\ \vec{eqn}}$$

$\boxed{\mathcal{H} \vdash node : \omega}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash \vec{eqn} \quad \mathcal{C} \vdash \vec{x} : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{H} \vdash \textbf{node} f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn} : \forall \bullet\ .\ (\vec{x} : ck) \to (\vec{y}' : ck')}$$

$\boxed{\mathcal{H} \vdash program}$

$$\frac{}{\emptyset \vdash \emptyset} \qquad \frac{\mathcal{H} \vdash \textbf{node} f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn} : \omega \quad (f : \omega) \uplus \mathcal{H} \vdash \vec{nodes}}{\mathcal{H} \vdash \textbf{node} f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn};;\vec{nodes}}$$

FIGURE 3.10 – Clocking rules for equations and nodes

The application of an OCaml function is well-clocked if *all* of its parameters have the same clock type. The clock of the result is also identical to the clock of the parameters :

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e_0 : ck \quad \mathcal{C} \vdash e_1 : ck \quad \dots \quad \mathcal{C} \vdash e_{n-1} : ck}{\mathcal{H}, \mathcal{C} \vdash y = \textbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1}}$$

A node is well-clocked provided that the equations it contains, as well as its input and output parameters, are all well-clocked. Its signature then corresponds to a type scheme quantified by the base clock of the node. It is an « arrow » type from the node's input parameters to its output parameters, all annotated with their clock types :

$$\frac{\mathcal{H}, \mathcal{C} \vdash \vec{eqn} \quad \mathcal{C} \vdash \vec{x} : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{H} \vdash \textbf{node} f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn} : \forall \bullet\ .\ (\vec{x} : ck) \to (\vec{y}' : ck')}$$

Finally, a program is well-clocked if all of its nodes are well-clocked :

$$\frac{}{\emptyset \vdash \emptyset} \qquad \frac{\mathcal{H} \vdash \textbf{node} f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn} : \omega \quad (f : \omega) \uplus \mathcal{H} \vdash \vec{nodes}}{\mathcal{H} \vdash \textbf{node} f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn};;\vec{nodes}}$$

A program that respects these various typing rules is guaranteed not to access, during its execution, values of absent flows ($\perp$), and thus not to exhibit unpredictable or erroneous behavior. In Section 5.2, we will detail the process of automatically verifying these various rules at the time of compiling an OCaLustre program.

3.2.5 Operational Semantics

The operational semantics of the OCaLustre language is similar to that of the Lustre language as defined in [CPHP87]. However, this reference semantics assumes the *inlining* of nodes, whereas we will see in Chapter 4 that the OCaLustre compiler follows a model of separate compilation. Although this compilation model does not accept certain synchronous programs that are valid in Lustre (we will address this limitation in Section 4.2.2), those it does accept do indeed conform to the semantics described in this section. This semantics is presented briefly, in the form of inference rules applied to a simplified representation of the normal form of OCaLustre programs. In this simplified representation, a program is a list of all the equations evaluated during a single execution instant. This amounts to saying that all node calls are replaced by their bodies, and that the integration (or *inlining*) of node calls is therefore performed, as in the Lustre compilation model. Furthermore, the equations of the program are represented annotated with their respective clocks, placed below the equality symbol. In addition, every constant is also annotated with its clock, placed as a superscript.

Let σ be the *memory* associated with an OCaLustre program, that is, a function that associates to each variable name a value at each execution instant. A memory σ is said to be *compatible* with the system of equations of a program if, for every equation $x = e$ in the system, the value of the right-hand side of the equation is identical to $\sigma(x)$ whenever the clock of x is true (otherwise, $\sigma(x) = \perp$ regardless of the value of e). In other words, a memory is compatible with the system of equations of the program if it is a solution to the latter.

We then define the following judgments :

— $\sigma \vdash exp : v$ means that, in a synchronous instant, the expression exp is reduced to the value v when evaluated in the context of the memory σ.

— $eqn \xrightarrow{\sigma} eqn'$ means that the equation eqn is compatible with the memory σ, and that it will be replaced by the expression eqn' for the execution of the next instant.

— $\vec{eqn} \xrightarrow{\sigma} \vec{eqn}'$ means that the list of equations $\vec{eqn}$ is compatible with the memory σ, and that it will be replaced by the list of equations $\vec{eqn}'$ for the execution of the next instant.

— Finally, $h \vdash \vec{eqn} : h'$ means that, given the history of inputs h of the program, the system of equations $\vec{eqn}$ of the program produces the history of outputs h'.

The full set of operational semantics rules of the language is described in Figure 3.11.

We describe here the main rules of the operational semantics :

— The value of a variable is extracted from the memory σ :

$$\frac{\sigma(x) = v}{\sigma \vdash x : v}$$

— If expression e_1 is reduced to the value v_1, and if expression e_2 is reduced to the value v_2, then the application of a binary infix operator to these two expressions $e_1 \diamond e_2$ is reduced to the value

$$\boxed{\sigma \vdash e : v}$$

$$\frac{\sigma(ck) = true}{\sigma \vdash ()^{ck} : ()} \qquad \frac{\sigma(ck) \neq true}{\sigma \vdash ()^{ck} : \bot} \qquad \frac{\sigma(ck) = true}{\sigma \vdash k^{ck} : k} \qquad \frac{\sigma(ck) \neq true}{\sigma \vdash k^{ck} : \bot}$$

$$\frac{\sigma(ck) = true}{\sigma \vdash X_i^{ck} : X_i} \qquad \frac{\sigma(ck) \neq true}{\sigma \vdash X_i^{ck} : \bot} \qquad \frac{\sigma(x) = v}{\sigma \vdash x : v} \qquad \frac{\sigma \vdash e : v}{\sigma \vdash \Box e : \Box v}$$

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1 \diamond e_2 : v_1 \diamond v_2} \qquad \frac{\sigma(x) = true \quad \sigma \vdash e : v}{\sigma \vdash e\, \mathbf{when}\, x : v} \qquad \frac{\sigma(x) \neq true}{\sigma \vdash e\, \mathbf{when}\, x : \bot}$$

$$\frac{\sigma(x) = false \quad \sigma \vdash e : v}{\sigma \vdash e\, \mathbf{whennot}\, x : v} \qquad \frac{\sigma(x) \neq false}{\sigma \vdash e\, \mathbf{whennot}\, x : \bot}$$

$$\boxed{\sigma \vdash ce : v}$$

$$\frac{\sigma \vdash e_1 : true \quad \sigma \vdash ce_2 : v \quad \sigma \vdash ce_3 : w}{\sigma \vdash \mathbf{if}\, e_1\, \mathbf{then}\, ce_2\, \mathbf{else}\, ce_3 : v} \qquad \frac{\sigma \vdash e_1 : false \quad \sigma \vdash ce_2 : v \quad \sigma \vdash ce_3 : w}{\sigma \vdash \mathbf{if}\, e_1\, \mathbf{then}\, ce_2\, \mathbf{else}\, ce_3 : w}$$

$$\frac{\sigma \vdash x : true \quad \sigma \vdash ce_1 : v \quad \sigma \vdash ce_2 : \bot}{\sigma \vdash \mathbf{merge}\, x\, ce_1\, ce_2 : v} \qquad \frac{\sigma \vdash x : false \quad \sigma \vdash ce_1 : \bot \quad \sigma \vdash ce_2 : w}{\sigma \vdash \mathbf{merge}\, x\, ce_1\, ce_2 : w}$$

$$\boxed{eqn \xrightarrow{\sigma} eqn'}$$

$$\frac{\sigma(ck) = true \quad \sigma(y) = k \quad \sigma \vdash e : k'}{y \underset{ck}{=} k\ \mathbf{fby}\ e \xrightarrow{\sigma} y \underset{ck}{=} k'\ \mathbf{fby}\ e} \qquad \frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} k\ \mathbf{fby}\ e \xrightarrow{\sigma} y \underset{ck}{=} k\ \mathbf{fby}\ e}$$

$$\frac{\sigma(ck) = true \quad \sigma(y) = w \quad \sigma \vdash e_0 : v_0 \quad \sigma \vdash e_1 : v_1 \quad \dots \quad \sigma \vdash e_{n-1} : v_{n-1} \quad (\mathtt{f}\ v_0\ v_1 \dots v_{n-1}) \Downarrow w}{y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1} \xrightarrow{\sigma} y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1}}$$

$$\frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1} \xrightarrow{\sigma} y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1}}$$

$$\frac{\sigma(ck) = true \quad \sigma(y) = v \quad \sigma \vdash ce : v}{y \underset{ck}{=} ce \xrightarrow{\sigma} y \underset{ck}{=} ce} \qquad \frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} ce \xrightarrow{\sigma} y \underset{ck}{=} ce}$$

$$\boxed{\vec{eqn} \xrightarrow{\sigma} \vec{eqn}'}$$

$$\frac{eqn \xrightarrow{\sigma} eqn' \quad \vec{eqn} \xrightarrow{\sigma} \vec{eqn}'}{eqn;\ \vec{eqn} \xrightarrow{\sigma} eqn';\ \vec{eqn}'}$$

$$\boxed{h \vdash \vec{eqn} : h'}$$

$$\frac{\vec{eqn} \xrightarrow{\sigma} \vec{eqn}' \quad h \vdash \vec{eqn}' : h'}{\sigma[input].h \vdash \vec{eqn} : \sigma[output].h'}$$

FIGURE 3.11 – Operational semantics of OCaLustre

$v_1 \diamond v_2$:

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1 \diamond e_2 : v_1 \diamond v_2}$$

— The result of the merge (**merge**) of two flows corresponds to the value of its second (resp. third) parameter when its first parameter is true (resp. false) :

$$\frac{\sigma \vdash x : true \quad \sigma \vdash ce_1 : v \quad \sigma \vdash ce_2 : \bot}{\sigma \vdash \mathbf{merge}\, x\, ce_1\, ce_2 : v} \qquad \frac{\sigma \vdash x : false \quad \sigma \vdash ce_1 : \bot \quad \sigma \vdash ce_2 : w}{\sigma \vdash \mathbf{merge}\, x\, ce_1\, ce_2 : w}$$

— A flow defined by an equation has no value ($\bot$) if its clock is false (or if it is itself absent) :

$$\frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} ce \xrightarrow{\sigma} y \underset{ck}{=} ce}$$

— The evaluation of an equation containing the **fby** operator implies a transformation of the equation's form for the next instant. The value on the left of the operator then corresponds to the value computed at the previous instant :

$$\frac{\sigma(ck) = true \quad \sigma(y) = k \quad \sigma \vdash e : k'}{y \underset{ck}{=} k\ \mathbf{fby}\ e \xrightarrow{\sigma} y \underset{ck}{=} k'\ \mathbf{fby}\ e}$$

— Evaluating an OCaml function call (designated by **call**) amounts to evaluating all the parameters of the call (e_0 through e_n) in OCaLustre, then applying the function to the computed values. The result w of this application, computed according to the semantics of OCaml[11], corresponds to the value of the resulting flow y.

$$\frac{\sigma(ck) = true \quad \sigma(y) = w \quad \sigma \vdash e_0 : v_0 \quad \sigma \vdash e_1 : v_1 \quad \dots \quad \sigma \vdash e_{n-1} : v_{n-1} \quad (\mathtt{f}\, v_0\, v_1 \dots v_{n-1}) \Downarrow w}{y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1} \xrightarrow{\sigma} y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \dots e_{n-1}}$$

— Finally, evaluating a program amounts to evaluating the equations it contains by computing a solution to the system of equations for given input values. It then produces a set of values corresponding to the program's output variables, and recursively induces the evaluation of new outputs from the inputs of the next instant.

$$\frac{\vec{eqn} \xrightarrow{\sigma} \vec{eqn}' \quad h \vdash \vec{eqn}' : h'}{\sigma[input].h \vdash \vec{eqn} : \sigma[output].h'}$$

To illustrate these various semantic rules, Figure 3.12 shows a derivation of the evaluation of the equation $y \underset{\bullet}{=} 2\ \mathbf{fby}\ (y+1)$: at the next execution instant, this equation becomes $y \underset{\bullet}{=} 3\ \mathbf{fby}\ (y+1)$.

A memory is *consistent* from the clocking point of view if and only if it associates $\bot$ with every variable whose value is not computed (i.e. if its clock is false or if this clock itself is not computed). An expected property of this semantics is that if the program is well-scheduled, then a memory compatible with this program is necessarily consistent (therefore $\bot$ is never consulted during the evaluation of expressions in

11. The notation $e \Downarrow w$ indicates that in the semantics of OCaml the expression e evaluates to the value v

$$\dfrac{\sigma(\bullet) = \mathit{true} \qquad \sigma(y) = 2 \qquad \dfrac{\dfrac{\sigma(y) = 2}{\sigma \vdash y : 2} \quad \dfrac{}{\sigma \vdash 1 : 1}}{\sigma \vdash (y+1) : 3}}{y \underset{\bullet}{=} 2\ \mathbf{fby}\ (y+1) \xrightarrow{\sigma} y \underset{\bullet}{=} 3\ \mathbf{fby}\ (y+1)}$$

FIGURE 3.12 – Derivation of the evaluation of an OCaLustre flow

a well-scheduled program). The proof of this property, which establishes the link between typing and semantics, is however deferred to future work, which could draw on techniques similar to those used in related work on the certification of Lustre compilation [Aug13, BBD^{+}17]. On the other hand, with the semantics described in this section there may exist multiple solutions to the system of equations of a synchronous instant. Additional static constraints are necessary to guarantee the existence of a unique solution, and to allow the sequential computation of this solution. In the next chapter, we will detail the nature of these causality and schedulability properties, which form the basis of the single-loop compilation method.

Chapter Conclusion

This chapter has provided a complete overview of the constructs and principles offered by our synchronous extension of the OCaml language. We have addressed many formal aspects, in particular the typing properties inherent to the use of this programming paradigm. OCaLustre is intended to be used together with the OMicroB virtual machine presented in the previous chapter, and it is therefore essential that it be fully compatible with the OMicroB bytecode interpreter. This is also the case for Lucid Synchrone, but its high expressiveness (higher-order features, multiple base clocks for the same node, automata, . . .) produces programs that consume more resources, which are not compatible with the hardware limitations we consider. On this subject, we will present in Chapter 7 some comparative elements that illustrate how our solution helps to limit the memory footprint of synchronous programs, since the OCaLustre compilation model is simplified by its more restricted expressiveness.

In the following chapter, we then present the steps that allow the compilation of an OCaLustre program into a standard sequential OCaml program, fully compatible with OMicroB and the targeted devices, but also with any OCaml compiler.

4 Compilation of OCaLustre Programs

OCaLustre is an extension of the OCaml language, and as such we use a set of software tools that enable the definition of such extensions. In particular, we make use of *PPX* [⚓3], a tool of the standard OCaml compiler that allows the definition of syntactic extensions through the use of specific annotations in the program's source code. In our case, we annotate what is initially considered by the OCaml compiler to be a function definition with the keyword « `node` » in order to construct an OCaLustre node (hence the syntax of annotations of the form `let%node`).

The advantage of using PPX is to delegate to the standard OCaml compiler[1] the task of performing the necessary steps of parsing the source program and constructing an internal representation within the compiler. The OCaLustre compiler can then plug directly into this internal representation (an abstract syntax tree — « *AST* » — in OCaml), apply several transformations to it, and then hand it back to the standard compiler, which is then responsible for producing executable code. The OCaLustre compiler can thus be regarded as a *preprocessor*, responsible for modifying the structure of the program's AST before it is further handled by the standard compiler.

In this chapter, we describe the steps required to compile an OCaLustre program, from its normalization into normal form to the generation of a standard representation of an OCaml program. We follow a process well-documented in the literature [Pla88, BCHP08], which has in the past been used to implement compilers for synchronous languages comparable to OCaLustre.

Figure 4.1 schematically represents the compilation chain of an OCaLustre program. It consists of four main steps :

— A first step of *normalization* transforms the program by restructuring its components in order to simplify the static analyses applied during its compilation.

— A second step of *scheduling* consists in sorting the various equations that constitute the body of an OCaLustre node so that their reading order corresponds to their declaration order during the generation of OCaml code.

— The third compilation step consists in the *inference of clock types* for OCaLustre equations. This step automatically annotates each equation of a program with its clock, following the clock typing rules defined in Section 3.2.4.

— The final step of compiling an OCaLustre program consists in the *translation* of the annotated OCaLustre program into a standard OCaml program, in the form of an AST understandable by the compiler of the language. This AST is then processed by the latter and compiled (in the case of a compilation into a bytecode file) into a file fully compatible with any OCaml virtual machine, including OMicroB.

It should be noted that our decision to follow a model of *separate compilation* of OCaLustre nodes (similar to the model of SCADE, as opposed to that of Lustre V4 [HR01], which performs *inlining* of calls) is based on the desire to avoid code duplication when compiling multiple calls to the same node.

1. Whether it be *ocamlc*, the compiler that generates OCaml bytecode, or *ocamlopt*, the compiler that generates native code.

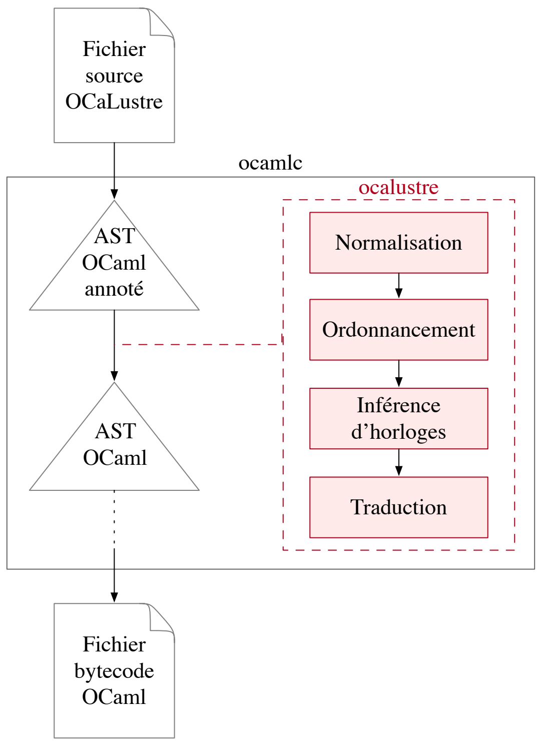


FIGURE 4.1 – Compilation chain of an OCaLustre program

Given the limited memory resources of a microcontroller, it is indeed important in our use case to reduce the memory footprint of programs. We will see during the description of the scheduling process that this decision has consequences for the static validation of certain programs, but this disadvantage seems acceptable to us given the importance, in our work, of limiting the resource consumption of programs.

4.1 Normalization into Normal Form

The first step in compiling an OCaLustre program consists in transforming the nodes resulting from the syntactic and lexical analysis of the source program into their representation in the *normal form* defined in Section 3.2.1.

The main role of normalization, or *conversion into normal form*, is to extract, within the equations of a node, the expressions that we call "*side-effect expressions*" in order to simulate, in the target language (OCaml), the use of OCaLustre's strict-evaluation conditional operator. This transformation thus makes it possible to preserve the semantics of OCaLustre without having to define a special « **`if`** » operator dedicated to the execution of OCaLustre nodes. Side-effect expressions are those that, during their evaluation, affect the internal state of a node. In OCaLustre, there are three of them :

1. The shift operator $\ggg$ implicitly involves the manipulation of a register in the internal state of the node in which it is used : the expression `0` $\ggg$ `e` is therefore a side-effect expression that induces both the computation of the value of the expression `e` at instant t, and the storage of this value in the internal state of the node so that it can be accessed by the program at instant $t+1$.

   To ensure that this evaluation is not "frozen" in the case where this expression is nested within the consequence or the alternative branch of a conditional, any sub-expression appearing in the definition of a flow `x` that contains a $\ggg$ is extracted from the expression in which it was nested and moved into the definition of a new flow, named `x_aux`. In this way, the value of `x_aux` is effectively computed at every execution instant of the program.

For example, the following non-normalized node :

```
let%node ex_norm (b) ~return:(x) =
  x = if b then (0 >>> (x + 1)) else (0 >>> x)
```

is translated into normal form as the following semantically equivalent representation :

```
node ex_norm b return x =
   x_aux1 = 0 fby (x + 1);
   x_aux2 = 0 fby x;
   x = if b then x_aux1 else x_aux2
```

Thus, the computation of `0 >>> (x + 1)` and of `0 >>> x` is indeed carried out at each execution instant (whatever the value of `b`), without having to distinguish between the lazy-evaluation `if` of the OCaml language and the strict-evaluation `if` of OCaLustre.

2. In the same way, since the body of a node may contain equations using the shift operator, any call to the latter can produce the same kind of side effects. The extraction mechanism is therefore also extended to applications. For example, the following node :

```
let%node call_count_cond (b) ~return:(x) =
  x = if b then count () else 0
```

becomes, after normalization :

```
node ex_norm b return x =
   x_aux = count ();
   x = if b then x_aux else 0
```

3. Finally, the same operation as for node calls is applied to each use of the `call` operator, since the called function may itself perform side effects.

More generally, normalization consists in transforming an OCaLustre program from a representation corresponding to the concrete syntax of the language into an AST, defined from the normalized grammar described in Section 3.2.1, which can be manipulated by the compiler. This process standardizes the internal representation of nodes, as well as their analysis, by structuring the various types of expressions (control expressions, normal expressions, . . .) and simplifying certain constructs. For example, tuples present in the concrete syntax of the language are broken down into several distinct equations after normalization. The following program :

```
let%node ex_tuples (a,b) ~return:(c,d) =
  (c,d) = (a,b)
```

is represented, in normal form, as follows :

```
node ex_tuples (a,b) return (c,d) =
  c = a;
  d = b
```

The normalization of an OCaLustre program corresponds to the application of a function *normalize* that transforms the abstract syntax tree resulting from the syntactic and lexical analysis of an OCaLustre source program into an AST corresponding to the program's internal, normalized representation (ast_{norm}). The normalization function can be decomposed into two functions : a first function *split* that "breaks down" tuples (given in Figure 4.2), followed by the function *norm* which actually performs normalization by distributing expressions into their equivalent in the grammar of the normal form. The normalization function *norm* for a node is illustrated in Figure 4.3 ; it calls mutually recursive auxiliary functions $norm_x$, where the index x corresponds to the name of the syntactic category into which each function converts a given expression or equation. The role of these functions is to traverse the AST of the program obtained from its syntactic analysis (with the source code represented in roman type) in order to produce an AST in normalized form, while collecting new equations resulting from the reorganization of the AST according to the normal form. Whenever a new equation must be created, they call a function $fresh()$ that generates a fresh variable name at each call.

$$
\begin{aligned}
split(\vec{y} \texttt{=} f\ e) &= [\vec{y} \texttt{=} f\ e] \\
split(y \texttt{=} e) &= [y \texttt{=} e] \\
split(y \texttt{,} \vec{y} \texttt{=} e \texttt{,} \vec{e}) &= (y \texttt{=} e) :: split(\vec{y} \texttt{=} \vec{e}) \\
split(eqn_1 \texttt{;} \vec{eqn}_2) &= let\ \vec{eqn}'_1 = split(eqn_1)\ in \\
&\quad\ let\ \vec{eqn}_2{}' = split(\vec{eqn}_2)\ in \\
&\quad\ \vec{eqn}'_1 +\!\!+ \vec{eqn}'_2 \\
split(\texttt{let\%node}\ f\ \vec{x}\ \texttt{\textasciitilde return:}\vec{y} \texttt{=} \vec{eqn}) &= let\ \vec{eqn}' = split(\vec{eqn})\ in \\
&\quad\ \texttt{let\%node}\ f\ \vec{x}\ \texttt{\textasciitilde return:}\vec{y} \texttt{=} \vec{eqn}'
\end{aligned}
$$

FIGURE 4.2 – Tuple-splitting function

4.2 Scheduling

In an OCaLustre program, the various equations constituting the body of a node are not subject to any constraint governing their order of appearance. Indeed, since the body of a node is a system of equations whose values are computed at each instant, nothing prevents (as in a classical mathematical system of equations) one of these equations from referring to a variable defined several lines later in the natural reading order (top to bottom). As a result, the following node is a perfectly valid example, even though the equation defining the flow `z` refers to the flow `k`, which has not yet been defined in the natural reading order, and the equation defining the flow `k` refers to `w` before it too has been defined :

$$
\begin{aligned}
norm_e(e_1 \diamond e_2) &= let\ (l_1, e_1') = norm_e(e_1)\ in\\
&\quad let\ (l_2, e_2') = norm_e(e_2)\ in\ (l_1 +\!\!+\, l_2, e_1' \diamond e_2')\\
norm_e(\square\ e_0) &= let\ (l, e_0') = norm_e(e_0)\ in\ (l, \square\ e_0')\\
norm_e(e_0\ \texttt{[@when}\ x\texttt{]}) &= let\ (l, e_0') = norm_e(e_0)\ in\ (l, e_0'\ \textbf{when}\ x)\\
norm_e(e_0\ \texttt{[@whennot}\ x\texttt{]}) &= let\ (l, e_0') = norm_e(e)\ in\ (l, e_0'\ \textbf{whennot}\ x)\\
norm_e(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= let\ x = fresh()\ in\\
&\quad let\ (l, eqn) = norm_{eqn}(x\ \texttt{=}\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)\ in\ (eqn :: l, x)\\
norm_e(\texttt{merge}\ x\ e_1\ e_2) &= let\ x = fresh()\ in\\
&\quad let\ (l, eqn) = norm_{eqn}(x\ \texttt{=}\ \texttt{merge}\ x\ e_1\ e_2)\ in\ (eqn :: l, x)\\
norm_e(k\ \texttt{->>}\ e) &= let\ x = fresh()\ in\\
&\quad let\ (l, eqn) = norm_{eqn}(x\ \texttt{=}\ k\ \texttt{->>}\ e)\ in\ (eqn :: l, x)\\
norm_e(\texttt{call}\ \texttt{f}\ e_0\ e_1\ \ldots\ e_{n-1}) &= let\ x = fresh()\ in\\
&\quad let\ (l, eqn) = norm_{eqn}(x\ \texttt{=}\ \texttt{call}\ \texttt{f}\ e_0\ e_1\ \ldots\ e_{n-1})\ in\ (eqn :: l, x)\\
norm_e(f\ \texttt{(}e_0\texttt{)}) &= let\ x = fresh()\ in\\
&\quad let\ (l, eqn) = norm_{eqn}(x = f\,\texttt{(}e_0\texttt{)})\ in\ (eqn :: l, x)\\
norm_e(e_0) &= ([], e_0)\ \text{pour toutes les autres formes de}\ e_0
\end{aligned}
$$

$$
\begin{aligned}
norm_{ce}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= let\ (l_1, e_1') = norm_e(e_1)\ in\\
&\quad let\ (l_2, ce_2) = norm_{ce}(e_2)\ in\\
&\quad let\ (l_3, ce_3) = norm_{ce}(e_3)\ in\\
&\quad (l_1 +\!\!+\, l_2 +\!\!+\, l_3, \textbf{if}\ e_1'\ \textbf{then}\ ce_2\ \textbf{else}\ ce_3)\\
norm_{ce}(\texttt{merge}\ x\ e_1\ e_2) &= let\ (l_1, e_1') = norm_e(e_1)\ in\\
&\quad let\ (l_2, e_2') = norm_e(e_2)\ in\\
&\quad (l_1 +\!\!+\, l_2, \textbf{merge}\ x\ e_1'\ e_2')\\
norm_{ce}(e_0) &= norm_e(e_0)\ \text{pour toutes les autres formes de}\ e_0
\end{aligned}
$$

$$
\begin{aligned}
norm_{\vec{e}}(e_0) &= norm_e(e_0)\\
norm_{\vec{e}}(e_1\,\texttt{,}\,e_2) &= let\ (l_1, e_1') = norm_e(e_1)\ in\\
&\quad let\ (l_2, \vec{e_2}) = norm_{\vec{e}}(e_2)\ in\ (l_1 +\!\!+\, l_2, e_1' :: \vec{e_2})
\end{aligned}
$$

$$
\begin{aligned}
norm_{eqn}(y\ \texttt{=}\ k\ \texttt{->>}\ e) &= let\ (l, e') = norm_e(e)\ in\ (l, y = k\ \textbf{fby}\ e')\\
norm_{eqn}(\vec{y}\ \texttt{=}\ f\,\texttt{(}e\texttt{)}) &= let\ (l, \vec{e}) = norm_{\vec{e}}(e)\ in\ (l, \vec{y} = f\ (\vec{e}))\\
norm_{eqn}(y\ \texttt{=}\ \texttt{call}\ \texttt{f}\ e_0\ e_1\ \ldots\ e_{n-1}) &= let\ (l_0, e_0') = norm_e(e_0)\ in\\
&\quad let\ (l_1, e_1') = norm_e(e_1)\ in \ldots\\
&\quad let\ (l_{n-1}, e_{n-1}') = norm_e(e_{n-1})\ in\ (l_0 +\!\!+\, l_1 +\!\!+\, \ldots +\!\!+\, l_{n-1}, y = \textbf{call}\ \texttt{f}\ e_0'\ e_1'\ \ldots\ e_{n-1}')\\
norm_{eqn}(y\ \texttt{=}\ e) &= let\ (l, ce) = norm_{ce}(e)\ in\ (l, y = ce)
\end{aligned}
$$

$$
\begin{aligned}
norm_{\vec{eqn}}(eqn) &= let\ (l, eqn) = norm_{eqn}(eqn)\ in\ (l, [eqn])\\
norm_{\vec{eqn}}(eqn\,\texttt{;}\,\vec{eqn}) &= let\ (l, eqn') = norm_{eqn}(eqn)\ in\\
&\quad let\ (l', \vec{eqn}') = norm_{\vec{eqn}}(\vec{eqn})\ in\ (l +\!\!+\, l', eqn' :: \vec{eqn}')
\end{aligned}
$$

$$
norm(\texttt{let\%node}\ f\ \vec{x}\ \texttt{\textasciitilde return:}\,\vec{y}\ \texttt{=}\ \vec{eqn}) = let\ (\vec{eqn}', \vec{eqn}'') = norm_{\vec{eqn}}(\vec{eqn})\ in\ \textbf{node}\ f\ \vec{x}\ \textbf{return}\ \vec{y} = (\vec{eqn}' +\!\!+\, \vec{eqn}'')
$$

Figure 4.3 – Normalization functions

```
node sched (x,y) return (z,k) =
   z = 3 * k;
   k = if  x then w else  4;
   w = y + 42
```

The OCaLustre compiler then performs, with the ultimate goal of transforming OCaLustre equations into OCaml variable declarations, a *scheduling* of the body of a node. This process consists in reorganizing the system of equations so that each flow definition appears before its use in other equations. Thus, the scheduling of the previous example consists in processing a version of `ordo` where each flow definition appears before its use :

```
node sched_correct (x,y) return (z,k) =
   w = y + 42;
   k = if  x then w else  4;
   z = 3 * k
```

Of course, the scheduling of the body of a node in no way modifies the semantics of the language, since each equation is considered to be computed *in parallel* with the other equations : the reading order of the equations does not reflect their actual execution semantics (which defines no sequential order of evaluation).

The implementation of equation scheduling is based on the creation of a directed acyclic graph[2] representing the dependencies between the various flows *within the same instant*. A flow y is considered a dependency of a flow x whenever y is referenced (within the same instant) in the definition of x. In the dependency graph, an edge from a vertex x to a vertex y represents the fact that the flow x depends on y. For example, Figure 4.4 shows the dependency graph of the node `sched`.

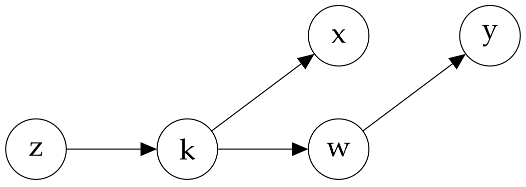


FIGURE 4.4 – Dependency graph of the node `sched`

Once this graph has been constructed, the scheduling of a node simply consists in following its reverse topological sort (without taking into account the input flows of the node) in order to produce the sequence of equations that constitutes the body of the node in question. In our example, the scheduling of `sched` therefore consists in defining first `w`, then `k`, and finally `z`.

An equation is well-scheduled provided that all the variables it contains, and on which it depends *instantaneously*, have been previously defined. Formally, we define the function *idents*, which returns the list of all variables present in an expression (or a list of expressions) :

2. The erroneous case in which this graph would be cyclic will be described in the following subsection.

$\boxed{idents(e)}$

$$\begin{aligned}
idents(k) &\equiv \varnothing\\
idents(x) &\equiv [x]\\
idents(X_i) &\equiv \varnothing\\
idents(e \diamond e') &\equiv idents(e) +\!\!+\, idents(e')\\
idents(\square\, e) &\equiv idents(e)\\
idents(e\ \mathbf{when}\ x) &\equiv x :: idents(e)\\
idents(e\ \mathbf{whennot}\ x) &\equiv x :: idents(e)
\end{aligned}$$

$\boxed{idents(\vec{e})}$

$$\begin{aligned}
idents([e]) &\equiv idents(e)\\
idents(e, \vec{e}) &\equiv idents(e) +\!\!+\, idents(\vec{e})
\end{aligned}$$

$\boxed{idents(ce)}$

$$\begin{aligned}
idents(e) &\equiv idents(e)\\
idents(\mathbf{merge}\ x\ ce\ ce') &\equiv x :: idents(ce) +\!\!+\, idents(ce')\\
idents(\mathbf{if}\ e\ \mathbf{then}\ ce'\ \mathbf{else}\ ce'') &\equiv idents(e) +\!\!+\, idents(ce') +\!\!+\, idents(ce'')
\end{aligned}$$

For any list of variables $\mathcal{V}$, the following rules define the notion of *well-scheduled* equations in the context where the variables contained in $\mathcal{V}$ have all been previously defined. We then write $\mathcal{V} \vdash_{ws} eqn$ to mean that the equation *eqn* is well-scheduled ("ws" for "well scheduled").

$\boxed{\mathcal{V} \vdash_{ws} eqn}$

$$\frac{y \notin \mathcal{V} \qquad idents(ce) \in \mathcal{V}}{\mathcal{V} \vdash_{ws} y = ce} \qquad \frac{y \notin \mathcal{V}}{\mathcal{V} \vdash_{ws} y = k\ \mathbf{fby}\ e}$$

$$\frac{\vec{y} \notin \mathcal{V} \qquad idents(\vec{e}) \in \mathcal{V}}{\mathcal{V} \vdash_{ws} \vec{y} = f(\vec{e})} \qquad \frac{y \notin \mathcal{V} \qquad idents(e_0) \in \mathcal{V} \qquad idents(e_1) \in \mathcal{V} \quad \ldots \quad idents(e_{n-1}) \in \mathcal{V}}{\mathcal{V} \vdash_{ws} y = \ \mathbf{call}\ \mathbf{f}\ e_0\ e_1 \ldots e_{n-1}}$$

$\boxed{\mathcal{V} \vdash_{ws} \vec{eqn}}$

$$\frac{\mathcal{V} \vdash_{ws} eqn}{\mathcal{V} \vdash_{ws} [eqn]} \qquad \frac{\mathcal{V} \vdash_{ws} y = ce \quad y :: \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} y = ce;\ \vec{eqn}}$$

$$\frac{\mathcal{V} \vdash_{ws} y = k\ \mathbf{fby}\ e \quad y :: \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} y = k\ \mathbf{fby}\ e;\ \vec{eqn}} \qquad \frac{\mathcal{V} \vdash_{ws} \vec{y} = f(\vec{e}) \quad \vec{y} +\!\!+ \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} \vec{y} = f(\vec{e});\ \vec{eqn}}$$

$$\frac{\mathcal{V} \vdash_{ws} y = \ \mathbf{call}\ \mathtt{f}\, e_0\, e_1 \ldots e_{n-1} \quad y :: \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} y = \ \mathbf{call}\ \mathtt{f}\, e_0\, e_1 \ldots e_{n-1};\ \vec{eqn}}$$

A node is then well-scheduled provided that all the equations that constitute its body are themselves well-scheduled, in the context where the variables $\vec{x}$ representing its input parameters are defined :

$\boxed{\vdash_{ws} node}$

$$\frac{\vec{x} \vdash_{ws} \vec{eqn}}{\vdash_{ws} \mathbf{node}\, f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn}}$$

The scheduling process of OCaLustre nodes is carried out after the normalization of the program. Scheduling is therefore performed on the intermediate representation of the program and can be seen as a function *schedule* of type $ast_{norm} \rightarrow ast_{norm}$.

For example, consider the following (normalized) program $\mathcal{P}$:

```
node f a return b =
  b = c + 1;
  c = 0 fby a

node g x return (y,z) =
  z = y + x;
  y = x
```

Then, the result of applying *schedule*($\mathcal{P}$) is :

```
node f a return b =
  c = 0 fby a;
  b = c + 1

node g x return (y,z) =
  y = x;
  z = y + x
```

4.2.1 Detection of Causality Loops

The scheduling step of OCaLustre nodes makes it possible to statically detect errors or inconsistencies in the definition of a node's flows. For example, the following node has an indeterminate semantics :

```
node causal_loop () return (c,d) =
  c = 5 * d;
  d = 2 − c
```

Indeed, in this example the flow c depends on the flow d, and the flow d in turn depends on c. The flows c and d thus depend on each other *within the same instant*. It is therefore impossible to compute, for example, the value of c, since it depends on that of d, which itself depends on c : there are therefore cyclic dependencies between these two flows.

This situation, called a *causality loop*, can appear in less obvious ways, within a long chain of dependencies between flows, and even across various calls to auxiliary nodes. For this reason, the OCaLustre compiler automatically detects such cyclic dependencies and rejects programs that contain them. This detection is simply performed by checking that no loop exists in the graph representing dependencies between flows : the presence of a cycle in this graph causes compilation to stop and an error message to be displayed :

```
Error: Causality loop in node "causal_loop" with variables (d, c)
```

It should also be noted that an equation such as `y = 0 ⋙ (y+1)` does not indicate that the flow y depends on itself, since the use of the ⋙ operator implies that it is the value of the flow y at the *previous* instant that is considered in the expression defining y, and dependency relations are only computed for the same instant. Thus, such an equation is fully compatible with the semantics of the language and does not indicate the existence of a causality loop.

4.2.2 Limitation Due to Separate Compilation

This scheduling process for the body of nodes, combined with our model of separate compilation of each OCaLustre node, can lead to the rejection of certain programs that are nevertheless correct. For example, consider the following program fragment :

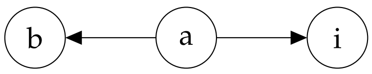
```
node shift_one x return y =
  y = 0 fby x

node call_shift_one i return b =
  a = b + i;
  b = shift_one a
```

During the scheduling of the node call_shift_one, the following dependency graph is generated :

$$\begin{tikzcd} b \arrow[r, bend right] & a \arrow[l, bend right] \arrow[r] & i \end{tikzcd}$$

Since this graph contains a loop, the node is therefore rejected at compilation. However, if we consider a version of the program in which *inlining* of `shift_one` is performed, that is, the replacement of the call to `shift_one` by the equation contained in its body, then a different dependency graph is generated :

```
node call_shift_one i return b =
  a = b + i;
  b = 0 fby a
```



In fact, in reality the flow *b* does not *instantaneously* depend on `a`, since `a` appears on the right-hand side of the ⋙ operator once the inlining of `shift_one` has been performed : the program did not contain a causality loop. The temporal shift information between the inputs and outputs of a node is not known by the OCaLustre compiler, which treats external nodes as black boxes, and thus assumes that every parameter of a call to a node is an instantaneous dependency for its output flows.

In our context of use, this limitation nevertheless seems acceptable, insofar as the separate compilation model avoids duplicating the body of a node for each equation that calls it, and thus helps reduce the final memory footprint of OCaLustre programs.

4.3 Clock Inference

Once normalization and scheduling have been performed, the next step in compiling an OCaLustre program consists in inferring the clocks of each expression contained in the bodies of its various synchronous nodes.

This inference mechanism is derived from the type inference system in the style of Hindley-Milner [Mil78], following the work of Colaço and Pouzet [CP03] on clock typing, while restricting polymorphism to a single type variable : the base clock (●).

Once a clock type has been assigned to each expression of a program, and thus the program satisfies the semantics of clock typing (3.2.4), the corresponding AST is annotated by associating each equation and each constant with its clock. These annotations are particularly necessary for the static verification of clock typing, which we will address in Section 5.2.

The abstract syntax tree generated in this way has a structure slightly different from that of a normalized program. It corresponds to the grammar of the normal form in which certain elements are annotated with their clock type. We denote by $ast_{clocked}$ an abstract syntax tree built from this grammar, and the clock inference process then corresponds to a function *clocks* of type $ast_{norm} \rightarrow ast_{clocked}$. In the following, we describe only the constructs that differ from the grammar of the normal form.

— Constants are annotated with their clock type, inferred by the compiler :

$$\begin{array}{lcl} e & ::= & \\ & | & k^{ck} \\ & | & (\ldots) \end{array}$$

— And equations are annotated with their clock, in particular an application is annotated with its conditional application clock :

$$\begin{array}{lcll} eqn & ::= & & \text{équation} \\ & | & y \underset{ck}{=} ce & \quad\text{expression} \\ & | & y \underset{ck}{=} k\ \mathbf{fby}\ e & \quad\text{fby} \\ & | & \vec{y} \underset{ck}{=} f(\vec{e}) & \quad\text{application} \\ & | & y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1\ ..\ e_{n-1} & \quad\text{application de fonction OCaml} \end{array}$$

For example, consider the following program $\mathcal{Q}$:

```
node g (a,b,c) return z =
  x = a when c;
  y = b + 2;
  z = merge c x y
```

Then, the result of applying $clocks(\mathcal{Q})$ is :

$$\begin{array}{l} \mathbf{node}\ \mathtt{g\ (a,b,c)}\ \mathbf{return}\ \mathtt{z} = \\ \quad \mathtt{x} \underset{(\bullet\ \mathbf{on}\ c)}{=} \mathtt{a}\ \mathbf{when}\ \mathtt{c}; \\ \quad \mathtt{y} \underset{(\bullet\ \mathbf{onnot}\ c)}{=} \mathtt{b} + 2^{(\bullet\ \mathbf{onnot}\ c)}; \\ \quad \mathtt{z} \underset{\bullet}{=} \mathbf{merge}\ \mathtt{c\ x\ y} \end{array}$$

The OCaLustre compiler also includes a *verifier* for the inferred clock types. This verifier, whose proof of correctness will be given in Chapter 5, ensures that for a given program, the types inferred by the OCaLustre compiler are consistent with the formal description of the type system given earlier. This verifier is enabled using the `-check_clocks` option of the compiler.

4.4 Translation to OCaml

The final step in compiling an OCaLustre program consists in translating the program, in its intermediate representation annotated with clocks, into an OCaml AST compatible with any OCaml compiler.

This *translation* step is carried out by separately converting each node of an OCaLustre program into an OCaml function, following the approach described by Biernacki *et al.* [BCHP08] and used for the compilation of SCADE 6 [CPP17] as well as other compilers of "Lustre-like" languages [GHKT14, MDLM18]. This separate compilation model, distinct from Lustre's compilation model where all components of a program are grouped into a single main loop, was chosen in order to reduce the size of programs in which multiple calls are made to the same nodes, and also to allow better modularization of the various components of a program.

Our solution follows Lustre's *single-loop* compilation model, but with separate compilation : each OCaLustre node is compiled into a distinct function, while the program's main node, which calls the generated functions, is executed in an infinite loop that reads the program inputs, executes the main node, and writes the computed outputs.

In this section, we describe *compilation schemes* representing the translation of the AST of an OCaLustre program into an OCaml program. The translation from the OCaLustre AST to OCaml code is based on the translation function $[\![\cdot]\!]$ of type $ast_{clocked} \rightarrow ast_{ocaml}$, which traverses the OCaLustre program and returns an abstract syntax tree corresponding to an OCaml program, manipulable by the standard OCaml compiler.

Traduction des expressions

OCaLustre expressions are translated directly, with the subsampling operators used for clock typing being erased :

$$
\begin{aligned}
[\![()]\!] &= \texttt{()} \\
[\![k]\!] &= \texttt{k} \\
[\![x]\!] &= \texttt{x} \\
[\![X_i]\!] &= \texttt{X}_i \\
[\![e_1 \diamond e_2]\!] &= [\![e_1]\!] \diamond [\![e_2]\!] \\
[\![\square\ e]\!] &= \square\ [\![e]\!] \\
[\![e \textbf{ when } x]\!] &= [\![e]\!] \\
[\![e \textbf{ whennot } x]\!] &= [\![e]\!]
\end{aligned}
$$

Lists of OCaLustre expressions become OCaml tuples of expressions :

$$
\begin{aligned}
[\![[e]]\!] &= [\![e]\!] \\
[\![e, es]\!] &= [\![e]\!] \texttt{,} [\![es]\!]
\end{aligned}
$$

The **merge** operator is translated in the same way as the conditional operator :

$$
\begin{aligned}
[\![\textbf{if } e \textbf{ then } ce_1 \textbf{ else } ce_2]\!] &= \texttt{if } [\![e]\!] \texttt{ then } [\![ce_1]\!] \texttt{ else } [\![ce_2]\!] \\
[\![\textbf{merge } x\ ce_1\ ce_2]\!] &= \texttt{if } [\![x]\!] \texttt{ then } [\![ce_1]\!] \texttt{ else } [\![ce_2]\!]
\end{aligned}
$$

Translation of a Node and Its Equations

An OCaLustre node is translated into an OCaml function that generates a closure whose environment contains the registers required for the use of the *followed-by* operator, as well as the various initializations of the instances of auxiliary nodes used in the original node :

$$\llbracket \textbf{node}\ f\ (\vec{x})\ \textbf{returns}\ (\vec{y}) =\ \vec{eqs} \rrbracket =$$
$$\texttt{let}\ \llbracket f \rrbracket\ \texttt{()}\ =\ \mathit{inits}(\vec{eqs}, \texttt{fun}\ \llbracket \vec{x} \rrbracket\ \texttt{->}\ \mathit{decls}(\vec{eqs}, \mathit{updates}(\vec{eqs}, \llbracket \vec{y} \rrbracket))_0)_0$$

Initializations : The function *inits* generates the variable declarations that make up the environment of the created closure. It is parameterized by a list of equations $\vec{eqs}$, an integer n (needed to distinguish between two instances of the same node), and a continuation K that represents the subsequent compilation steps.

Each use of the ⋙ operator results in the definition of a register (an OCaml reference) initialized by the value on the left-hand side of this operator :

$$\mathit{inits}(y \underset{ck}{=} k\ \textbf{fby}\ e; \vec{eqs}, K)_n\ =\ \texttt{let \_}\llbracket y \rrbracket\texttt{\_fby = ref}\ \llbracket k \rrbracket\ \texttt{in}\ \mathit{inits}(\vec{eqs}, K)_n$$

Each call to an auxiliary node results in the generation of an instance of this node, by invoking the function of the same name. These instances are numbered to support the case where multiple instances of the same node are called :

$$\mathit{inits}(\vec{y} \underset{ck}{=} g\ (\vec{e}); \vec{eqs}, K)_n\ =\ \texttt{let}\ \llbracket g \rrbracket\texttt{\_}\llbracket n \rrbracket\ =\ \llbracket g \rrbracket\ \texttt{()}\ \texttt{in}\ \mathit{inits}(\vec{eqs}, K)_{n+1}$$

The other forms of equations do not produce any declaration in the environment of the created closure :

$$\mathit{inits}(eq; \vec{eqs}, K)_n = \mathit{inits}(\vec{eqs}, K)_n$$
$$\mathit{inits}(\varnothing, K)_n = K$$

A list of variables $\vec{y}$ is translated into a tuple of variables, or into the value *unit* if the list is empty :

$$\llbracket () \rrbracket = \texttt{()}$$
$$\llbracket y \rrbracket = \texttt{y}$$
$$\llbracket y, \vec{y} \rrbracket = \llbracket y \rrbracket \texttt{,} \llbracket \vec{y} \rrbracket$$

Declarations : The body of the node is traversed by the function *decls* in order to convert each equation into a variable declaration in the closure returned by the generated function. An equation is "guarded" by the condition coming from its clock (the function *guard* is defined later in this section) :

$$\begin{aligned}
decls(y \underset{ck}{=} ce; \vec{eqs}, K)_n &= \texttt{let}\ [\![y]\!] \ = \ guard(ck, ce)_y\ \texttt{in}\ decls(\vec{eqs}, K)_n \\
decls(y \underset{ck}{=} k\ \textbf{fby}\ e; \vec{eqs}, K)_n &= \texttt{let}\ [\![y]\!] \ = \ guard(ck, \texttt{!\_}[\![y]\!]\texttt{\_fby})_y\ \texttt{in}\ decls(\vec{eqs}, K)_n \\
decls(y \underset{ck}{=} \textbf{call}\ \texttt{f}\ e_0\ e_1\ \ldots\ e_{m-1}); \vec{eqs}, K)_n &= \texttt{let}\ [\![y]\!] \ = \ guard(ck, \texttt{f}\ [\![e_0]\!]\ [\![e_1]\!] \ldots [\![e_{m-1}]\!])_y\ \texttt{in}\ decls(\vec{eqs}, K)_n \\
decls(\vec{y} \underset{ck}{=} g\ (\vec{e}); \vec{eqs}, K)_n &= \texttt{let}\ [\![\vec{y}]\!] \ = \ guard(ck, [\![g]\!]\texttt{\_}[\![n]\!]\ [\![\vec{e}]\!])_{\vec{y}}\ \texttt{in}\ decls(\vec{eqs}, K)_{n+1} \\
decls(\varnothing, K)_n &= K
\end{aligned}$$

When the clock of a node call is false, a placeholder value representing absence (i.e. the $\perp$ of the language semantics) must be assigned to the declared variable. Several solutions can be considered to represent such a generic value. For example, one might consider translating all flows of the OCaLustre language into values of type `option`[3], and thus assign the OCaml value `None` to an absent flow, but this solution causes a significant increase in the size of the generated program and leads to systematic memory allocations. Another alternative would be to use an OCaml compiler capable of handling *nullable* types [MV14], with the disadvantage of losing portability of the generated code. More simply, the current OCaLustre compiler replaces any absent value ($\perp$) with the value `Obj.magic ()`. This special value has the particular property of being considered well-typed regardless of its usage context (for example, the expression `Obj.magic () + 42` causes no problem for the compiler), and thus violates the assumptions that guarantee the type safety of a program. It is therefore important to ensure statically that at no point during the execution of the program is this value actually used, and that it is present only to satisfy the construction of an OCaml program where an expression must always have a value (even if here it is merely a default value that is never used).

The function *guard* therefore conditions certain expression evaluations and replaces them with an absence value in the case where their clock is false (or absent). It calls a function *bottom*, which thus generates a representation of absence with the correct number of elements :

$$\begin{aligned}
guard(\bullet, e)_{\vec{y}} &= e \\
guard(ck, e)_{\vec{y}} &= \texttt{if}\ [\![ck]\!]\ \texttt{then}\ e\ \texttt{else}\ bottom(\vec{y})
\end{aligned}$$

with

$$\begin{aligned}
[\![\bullet]\!] &= \texttt{true} \\
[\![ck\ \textbf{on}\ x]\!] &= \texttt{(}[\![ck]\!]\ \texttt{\&\&}\ [\![x]\!]\texttt{)} \\
[\![ck\ \textbf{onnot}\ x]\!] &= \texttt{(}[\![ck]\!]\ \texttt{\&\&}\ \texttt{not}\ [\![x]\!]\texttt{)}
\end{aligned}$$

and

3. The definition of this type is as follows : `type 'a option = None | Some of 'a`.

$$\begin{aligned} bottom(()) &= \texttt{()} \\ bottom(y) &= \texttt{Obj.magic ()} \\ bottom(y, \vec{y}) &= \texttt{Obj.magic (),} bottom(\vec{y}) \end{aligned}$$

Updates : Before returning the tuple of variables corresponding to the output flows of the node, the generated closure updates each register resulting from the use of ⋙ using the function *updates* :

$$\begin{aligned} updates(x \underset{ck}{=} k \textbf{ fby } e :: \vec{eqs}, K) &= guard(ck, [\![x]\!]\texttt{\_fby := } [\![e]\!]) \texttt{ ; } updates(\vec{eqs}, K) \\ updates(eq :: \vec{eqs}, K) &= updates(\vec{eqs}) \text{ (pour les autres formes de } eq) \\ updates(\varnothing, K) &= K \end{aligned}$$

Translation of a Program

Finally, generating the OCaml code of an OCaLustre program amounts to sequentially generating the code of each node it contains :

$$\begin{aligned} [\![node :: \vec{node}]\!] &= [\![node]\!]\texttt{;;} \\ &\quad\ [\![\vec{node}]\!] \\ [\![node :: \varnothing]\!] &= [\![node]\!] \end{aligned}$$

4.5 Example

Let us consider a simple OCaLustre program that illustrates all the compilation steps described above. This program consists of two nodes : the first node, `counting`, increments a counter at each instant, which is reset if its parameter is *true*. The second node, `counting_and`, counts the number of times its second and third parameters are both true, as long as its first parameter is false :

```
let%node counting (reset) ~return:(c) =
  c = if reset then 0 else 0 ⋙ (c + 1);

let%node counting_and (reset,b1,b2) ~return:(clk,c_sampled) =
  c_sampled = counting (reset [@when clk]);
  clk = (b1 && b2)
```

After normalization, the side-effect expression `(c + 1)` in `counting` is extracted from the equation `c` into a separate equation, `c_aux` :

```
node counting (reset) return (c) =
  c_aux = 0 fby (c + 1);
  c = if  reset then 0 else c_aux

node count_and (reset,b1,b2) return (clk,c_sampled) =
  c_sampled = count (reset when clk);
  clk = (b1 && b2)
```

The scheduling step reorganizes the body of `count_and` so that the flow `c` is defined before its use in `c_sampled` :

```
node counting (reset) return (c) =
  c_aux = 0 fby (c + 1);
  c = if  reset then 0 else c_aux

node counting_and (reset,b1,b2) return (clk,c_sampled) =
  clk = (b1 && b2);
  c_sampled = counting (reset when clk)
```

After clock inference, the equations and constants are annotated with their clock types :

```
node counting (reset) return (c) =
  c_aux = 0 fby (c + 1•);
        •
  c = if reset then 0• else c_aux
    •

node counting_and (reset, b1, b2) return (clk, c_sampled) =
  clk = (b1 && b2);
      •
  c_sampled     =    counting (reset when clk)
             • on clk
```

Finally, the translation into a standard OCaml program produces the definition of two functions, each corresponding to a node of the original program :

```
let counting () =
  let c_aux_fby = ref 0 in
  fun reset ->
    let c_aux = !cpt_aux_fby in
    let c = if reset then 0 else c_aux in
    c_aux_fby := (c + 1);
    c

let counting_and () =
  let counting1 = counting () in
  fun (reset,b1,b2) ->
    let clk = b1 && b2 in
    let c_sampled = if clk then count1 reset else Obj.magic () in
    (clk,c_sampled)
```

For each node of type $t_{in} \rightarrow t_{out}$, the type of the OCaml function generated at the end of this compilation process is $unit \rightarrow (t_{in} \rightarrow t_{out})$, which corresponds to the type of a function that returns an *instance* of the node in the form of an OCaml function. In the previous example, the type of count is $unit \rightarrow (bool \rightarrow int)$, and that of count_and is $unit \rightarrow ((bool * bool * bool) \rightarrow (bool * int))$.

Generation of the Runtime Loop :

The code responsible for reading inputs from the environment, executing a synchronous instant, and producing the outputs computed during that instant is called the *runtime loop* [Bou98]. The code of the main function of the OCaml program, which implements a minimal runtime loop allowing the previous synchronous program to be executed, then has the following form :

```
let () =
  (* initialization of the main node *)
  let main = counting_and () in
  while true do
   (* reading inputs *)
   let (reset,b1,b2) = input_counting_and () in
   (* execution of one step of the main node *)
   let (clk,c_sampled) = main (reset,b1,b2) in
   (* writing outputs *)
   output_counting_and (clk,c_sampled)
  done
```

This code can be generated automatically by using the -m option (followed by the name of the main node) of the OCaLustre compiler. This option also generates a stub code to be completed, which contains the input and output functions of the synchronous program.

Conclusion du chapitre

In this chapter, we have described the different steps that make it possible to transform an OCaLustre program into a sequential OCaml program. These transformations provide several typing and schedulability guarantees that make it possible to build programs with reinforced safety. Several properties related to these typing guarantees can then be verified in order to validate this increase in safety. In the next chapter, we will describe the formalization and the proof of such properties.

5 OCaLustre Properties Formalized and Proved with Coq

In this chapter, we describe several properties that derive from the formal specification of OCaLustre, and we formalize and prove them in Coq. These properties concern the type systems described in Section 3.2 (which govern the "standard" typing of data and clock typing), and allow us to check the consistency between these formal systems and the implementation in the OCaLustre compiler prototype. Verifying these properties makes it possible to guarantee the safety of certain aspects of the OCaLustre compiler, such as the correctness of OCaLustre value typing after their translation into OCaml, as well as the consistency of the clock types inferred by OCaLustre with the formal system described previously. In particular, the code of a *clock verifier*, obtained by extraction with Coq from the formal specification of the clocking rules of a program, is integrated into the OCaLustre compiler, which follows the steps described in the previous chapter.

5.1 Translation and Typing Correctness

In order to leave to the type-checker included in the standard OCaml compiler the responsibility of verifying that OCaLustre nodes are well typed, we present in this section a proof of the typing correctness of OCaLustre with respect to its translation.

This proof makes it possible to establish two properties that ensure that OCaLustre programs are well typed if and only if the generated OCaml code is itself well typed :

— A *preservation* property, which states that if the OCaLustre code is well typed, then its translation is also well typed[1].

— A *safety* property, which states that if the generated OCaml code is well typed, then the corresponding OCaLustre node is also well typed : no typing information is lost that could turn a badly typed OCaLustre program into well-typed OCaml code.

The metatheoretical proof of these properties avoids the need to add an *ad hoc* type-checker to OCaLustre, and instead relies on the strength of the typing mechanism of the native OCaml compiler. It is indeed the OCaml type-checker that, if it detects a typing error in the generated code, makes it possible to deduce that the original OCaLustre program is ill typed.

In this section, we progressively describe the main steps of this correctness proof. Our reasoning also relies only on nodes considered "well formed". We consider that a node *node* is well formed (denoted $\vdash_{wf}$ *node*) if all the names of the flows it defines are distinct, if all its parameters are distinct, and if no input parameter name is reused to define a new flow name in the body of the node :

1. This property can be seen as a form of *completeness* in the sense that no correct program is rejected : typing errors are not introduced by the translation.

$\boxed{\vdash_{wf} node}$

$$\frac{distinct(\vec{x}) \quad distinct(\vec{y}) \quad \vec{x} \cap \vec{y} = \emptyset \quad distinct(names(\vec{eqn})) \quad \vec{x} \cap names(\vec{eqn}) = \emptyset}{\vdash_{wf} \textbf{node}\, f(\vec{x})\ \textbf{returns}\ (\vec{y}) = \vec{eqn}}$$

avec

$\boxed{distinct(\vec{x})}$

$$\frac{}{distinct(\emptyset)} \qquad \frac{y \notin \vec{y} \quad distinct(\vec{y})}{distinct(y :: \vec{y})}$$

et

$\boxed{names(\vec{eqn})}$

$$\begin{aligned} &names(\emptyset) \equiv \emptyset \\ &names(x \underset{ck}{=} ce; \vec{eqn}) \equiv x :: names(\vec{eqn}) \\ &names(x \underset{ck}{=} k\ \textbf{fby}\ e; \vec{eqn}) \equiv x :: names(\vec{eqn}) \end{aligned}$$

The correctness proof with respect to typing of the translation presented in this section relies mainly on an equivalence theorem, which states that a well-formed OCaLustre node is well typed if and only if its translation is also well typed :

Theorem 5.1.1 (Typing Correctness of a Node)**.**

$$\forall\ node\ t,\ \vdash_{wf} node \Rightarrow (\ \vdash node : t \Leftrightarrow\ \vdash [\![node]\!] : unit\ \rightarrow t)$$

The proof of this equivalence makes it possible to establish the two properties of preservation and safety. The left-to-right implication ensures, by contraposition, that if the translation of a node is ill typed in OCaml, then the node is also ill typed in OCaLustre (*preservation*). The right-to-left implication states that if the translation of a node is well typed, then the node itself is well typed. By contraposition, if a node is ill typed, then its translation is also ill typed (*safety*).

All of the theorems and auxiliary lemmas required for this proof are available online, in the form of documentation and Coq files [⚓1]. It should be noted, however, that this proof is carried out on a simplified version of OCaLustre as well as of the OCaml language (which we call *"pseudo-ML"*) : in particular, we do not handle the parametric polymorphism used in OCaml, OCaLustre programs are represented as a single node (with no possibility of node calls[2]), and flows can only be of type *int* or *bool*. Moreover, OCaml function calls through the keyword `call` have not been formalized. These should not conflict with the correctness of the proof since their compilation is direct : a `call` generates a function application (and the types coincide). The addition of node calls is more complex, since it introduces new names into the generated code after compilation (as each call to a node leads to the creation of a closure instance in the generated code). The integration of these constructions into the proof of typing correctness of the translation is work currently in progress. The version of the proof presented in this manuscript and in the associated Coq files is therefore simplified.

2. In this sense, the OCaLustre programs treated here are close to Lustre programs, for which, during compilation, the code of the various intermediate nodes is inlined into the main node.

5.1.1 Partial Translation Using Simultaneous Declarations

In order to break down the typing correctness proof, we first prove the equivalence of typing between the OCaLustre program and the same program translated into the *pseudo-ML* language extended with a non-standard construct. This construct, of the form “*let ... with ... in ...*”, allows the simultaneous definition of variables.

For example, the following code snippet is correct in *pseudo-ML*, whereas it would not be in the OCaml language, which requires that multiple variable definitions (constructed with the keyword “*and*”[3]) not depend on one another :

```
let x = y with y = 2 in x
```

This intermediate construct is not intended to be executed; it simply serves to factorize the proofs of typing correctness (postponing the introduction of the notion of well-scheduling to a proof step described later), for which the final theorem will not involve such simultaneous declarations.

Typing Rules of *pseudo-ML*

The grammar of the pseudo-ML language, very close to that of OCaml, is given in Figure 5.1. The main difference between this grammar and that of OCaml is the addition of the simultaneous declaration construct “*let ... with ...*” described above.

It should also be noted that, in order to simplify the proof process (by ignoring name-shadowing phenomena), the namespace reserved for variables newly introduced by the translation and the namespace reserved for variables already present in the node declaration are distinct by construction : the typing environment $\mathcal{R}$ contains the set of pairs $(name, type)$ of the variables introduced by the compilation, while the environment Γ concerns the pairs $(name, type)$ of the other variables. In this case, the names contained in $\mathcal{R}$ (resulting from the translation of $\ggg$) all correspond to references.

3. Mutually recursive definitions are only possible with functional values in OCaml.

e	$::=$		expressions
	$\mid$	$\bot$	magic
	$\mid$	$()$	unit
	$\mid$	k	constante
	$\mid$	x	variable
	$\mid$	x_r	registre
	$\mid$	$\mathbf{ref}\ e$	référence
	$\mid$	$!e$	déréférencement
	$\mid$	$e := e'$	assignation
	$\mid$	$e \diamond e'$	opérateur binaire
	$\mid$	$\square\, e$	opérateur unaire
	$\mid$	$\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e''$	conditionnelle
	$\mid$	$e\ ;\ e'$	séquence
	$\mid$	$(\vec{e})$	n-uplet
	$\mid$	$\mathbf{fun}\ \vec{x} \rightarrow\ e$	fonction n-aire
	$\mid$	$\mathbf{let}\ \delta\ \mathbf{in}\ e'$	déclarations de variables
	$\mid$	$\mathbf{let}\ \delta_r\ \mathbf{in}\ e'$	déclarations de variables (nouveaux noms)
	$\mid$	$\mathbf{let}\ y = e\ \mathbf{in}\ e'$	déclaration de variable
	$\mid$	$\mathbf{let}\ y_r = e\ \mathbf{in}\ e'$	déclaration de variable (nouveau nom)
δ	$::=$		déclarations de variables simultanées
	$\mid$	$\varnothing$	
	$\mid$	$(y = e)\ \mathbf{with}\ \delta$	
δ_r	$::=$		déclarations de variables simultanées (nouveaux noms)
	$\mid$	$\varnothing$	
	$\mid$	$(y_r = e)\ \mathbf{with}\ \delta_r$	

FIGURE 5.1 – Grammaire de *pseudo-ML*

The typing rules for simultaneous declarations are then as follows :

$\boxed{\mathcal{R}, \Gamma \vdash \delta : \vec{t}}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \varnothing : \varnothing} \qquad \frac{\Gamma(x) = t \quad \mathcal{R}, \Gamma \vdash e : t \quad \mathcal{R}, \Gamma \vdash \delta : \vec{t}}{\mathcal{R}, \Gamma \vdash (x = e)\ \mathbf{with}\ \delta : t * \vec{t}}$$

$\boxed{\mathcal{R}, \Gamma \vdash \delta_r : \vec{t}}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \varnothing : \varnothing} \qquad \frac{\mathcal{R}(x) = t \quad \mathcal{R}, \Gamma \vdash e : t \quad \mathcal{R}, \Gamma \vdash \delta_r : \vec{t}}{\mathcal{R}, \Gamma \vdash (x_r = e)\ \mathbf{with}\ \delta_r : t * \vec{t}}$$

They state that for a set of simultaneous variable declarations to be well typed, each of its elements must be well typed in the *same* typing environment.

All the other typing rules of *pseudo-ML*, which do not differ significantly from those of a standard ML language, are given in Figure 5.2.

$\boxed{\Gamma \vdash \vec{x} : \vec{t}}$

$$\frac{}{\Gamma \vdash \varnothing : \varnothing} \qquad \frac{\Gamma(x) = t \qquad \Gamma \vdash \vec{x} : \vec{t}}{\Gamma \vdash x :: \vec{x} : t * \vec{t}}$$

$\boxed{\mathcal{R}, \Gamma \vdash \vec{e} : \vec{t}}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \varnothing : \varnothing} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t \qquad \mathcal{R}, \Gamma \vdash \vec{e} : \vec{t}}{\mathcal{R}, \Gamma \vdash e, \vec{e} : t * \vec{t}}$$

$\boxed{\mathcal{R}, \Gamma \vdash e : t}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \bot : t} \qquad \frac{}{\mathcal{R}, \Gamma \vdash () : \mathbf{unit}} \qquad \frac{}{\mathcal{R}, \Gamma \vdash \mathit{int\_literal} : \mathbf{int}} \qquad \frac{}{\mathcal{R}, \Gamma \vdash \mathit{bool\_literal} : \mathbf{bool}}$$

$$\frac{\Gamma(x) = t}{\mathcal{R}, \Gamma \vdash x : t} \qquad \frac{\mathcal{R}(x) = t}{\mathcal{R}, \Gamma \vdash x_r : t} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t}{\mathcal{R}, \Gamma \vdash \square e : t}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : \mathbf{int} \qquad \mathcal{R}, \Gamma \vdash e' : \mathbf{int}}{\mathcal{R}, \Gamma \vdash e \diamond_{int} e' : \mathbf{int}} \qquad \frac{\mathcal{R}, \Gamma \vdash e : \mathbf{bool} \qquad \mathcal{R}, \Gamma \vdash e' : \mathbf{bool}}{\mathcal{R}, \Gamma \vdash e \diamond_{bool} e' : \mathbf{bool}}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : t \qquad \mathcal{R}, \Gamma \vdash e' : t}{\mathcal{R}, \Gamma \vdash e \diamond_{comp} e' : \mathbf{bool}} \qquad \frac{\mathcal{R}, \Gamma \vdash e : \mathbf{bool} \qquad \mathcal{R}, \Gamma \vdash e' : t \qquad \mathcal{R}, \Gamma \vdash e'' : t}{\mathcal{R}, \Gamma \vdash \mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' : t}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : t}{\mathcal{R}, \Gamma \vdash \mathbf{ref}\, e : t\, \mathbf{ref}} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t\, \mathbf{ref}}{\mathcal{R}, \Gamma \vdash\, !e : t} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t\, \mathbf{ref} \qquad \mathcal{R}, \Gamma \vdash e' : t}{\mathcal{R}, \Gamma \vdash e := e' : \mathbf{unit}}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : \mathbf{unit} \qquad \mathcal{R}, \Gamma \vdash e' : t}{\mathcal{R}, \Gamma \vdash e\ ;\ e' : t} \qquad \frac{\mathcal{R}, \Gamma \vdash \vec{e} : \vec{t}}{\mathcal{R}, \Gamma \vdash (\vec{e}) : \vec{t}} \qquad \frac{\Gamma' = \left(\vec{x} : \vec{t}\right) \qquad \mathcal{R}, \Gamma' \cup \Gamma \vdash e : t'}{\mathcal{R}, \Gamma \vdash \mathbf{fun}\, \vec{x} \rightarrow\ e : \vec{t} \rightarrow t'}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : t \qquad \mathcal{R}, \{x : t\} \cup \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, x = e\, \mathbf{in}\, e' : t'} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t \qquad \{x : t\} \cup \mathcal{R}, \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, x_r = e\, \mathbf{in}\, e' : t'}$$

$$\frac{\mathit{names}(\delta) = \vec{y} \qquad \Gamma' = \left(\vec{y} : \vec{t}\right) \qquad \mathcal{R}, \Gamma' \cup \Gamma \vdash \delta : \vec{t} \qquad \mathcal{R}, \Gamma' \cup \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, \delta\, \mathbf{in}\, e' : t'}$$

$$\frac{\mathit{names}(\delta_r) = \vec{y} \qquad \mathcal{R}' = \left(\vec{y} : \vec{t}\right) \qquad \mathcal{R}' \cup \mathcal{R}, \Gamma \vdash \delta_r : \vec{t} \qquad \mathcal{R}' \cup \mathcal{R}, \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, \delta_r\, \mathbf{in}\, e' : t'}$$

FIGURE 5.2 – Typing Rules of *pseudo-ML*

Algorithmic Typing Rules of OCaLustre

To carry out the proof of the expected equivalence, we reason on the basis of *algorithmic* typing rules for OCaLustre, similar to the *declarative* typing rules discussed in Section 3.2.2. These rules are given in Figure 5.3. The typing rule of a node refers to the *names* function introduced earlier, whose role is to return the names of the different flows defined by a list of equations. It should also be noted that, in order to present a set of easily understandable rules, the typing rule for nodes here assumes that the nodes of an OCaLustre program contain no locally scoped flows : all flows defined in the body of a node appear in its outputs. This temporary limitation, which allows us to follow the progression of the Coq proof, will be lifted at the end of this section, where we extend our proof to nodes that return only a subset of the flows defined by the equations in the body of a node.

$\boxed{\Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash () : \mathbf{unit}} \qquad \frac{}{\Gamma \vdash \mathit{int\_literal} : \mathbf{int}} \qquad \frac{}{\Gamma \vdash \mathit{bool\_literal} : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \square\, e : t}$$

$$\frac{\Gamma \vdash e : \mathbf{int} \quad \Gamma \vdash e' : \mathbf{int}}{\Gamma \vdash e \diamond_{int} e' : \mathbf{int}} \qquad \frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash e' : \mathbf{bool}}{\Gamma \vdash e \diamond_{bool} e' : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t}{\Gamma \vdash e \diamond_{comp} e' : \mathbf{bool}}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : \mathbf{bool}}{\Gamma \vdash e\, \mathbf{when}\, x : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : \mathbf{bool}}{\Gamma \vdash e\, \mathbf{whennot}\, x : t}$$

$\boxed{\Gamma \vdash_{ce} ce : t}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash_{ce} e : t} \qquad \frac{\Gamma \vdash x : \mathbf{bool} \quad \Gamma \vdash_{ce} ce : t \quad \Gamma \vdash_{ce} ce' : t}{\Gamma \vdash_{ce} \mathbf{merge}\, x\, ce\, ce' : t} \qquad \frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash_{ce} ce : t \quad \Gamma \vdash_{ce} ce' : t}{\Gamma \vdash_{ce} \mathbf{if}\, e\, \mathbf{then}\, ce\, \mathbf{else}\, ce' : t}$$

$\boxed{\Gamma \vdash ck}$

$$\frac{}{\Gamma \vdash \bullet} \qquad \frac{\Gamma \vdash ck \quad \Gamma(x) = \mathbf{bool}}{\Gamma \vdash ck\, \mathbf{on}\, x} \qquad \frac{\Gamma \vdash ck \quad \Gamma(x) = \mathbf{bool}}{\Gamma \vdash ck\, \mathbf{onnot}\, x}$$

$\boxed{\Gamma \vdash eqn : t}$

$$\frac{\Gamma \vdash ck \quad \Gamma(y) = t \quad \Gamma \vdash_{ce} ce : t}{\Gamma \vdash y \underset{ck}{=} ce : t} \qquad \frac{\Gamma \vdash ck \quad \Gamma(y) = t \quad \Gamma \vdash e : t \quad \Gamma \vdash k : t}{\Gamma \vdash y \underset{ck}{=} k\, \mathbf{fby}\, e : t}$$

$\boxed{\Gamma \vdash \vec{eqn} : \vec{t}}$

$$\frac{}{\Gamma \vdash \varnothing : \varnothing} \qquad \frac{\Gamma \vdash eqn : t \quad \Gamma \vdash \vec{eqn} : \vec{t}}{\Gamma \vdash eqn; \vec{eqn} : t * \vec{t}}$$

$\boxed{\vdash node : t}$

$$\frac{\Gamma = \left(\vec{y} : \vec{t'}\right) \quad \Gamma' = \left(\vec{x} : \vec{t}\right) \quad \Gamma \cup \Gamma' \vdash \vec{eqn} : \vec{t'} \quad names(\vec{eqn}) = \vec{y}}{\vdash \mathbf{node}\, f(\vec{x})\, \mathbf{returns}\, (\vec{y}) = \vec{eqn} : \vec{t} \rightarrow \vec{t'}}$$

FIGURE 5.3 – *Algorithmic* Typing Rules of OCaLustre

Translation into *pseudo-ML*

The translation function $[\![\cdot]\!]$ from an OCaLustre node into *pseudo-ML* code is given in Figure 5.4. This translation is similar to that from OCaLustre to OCaml, except that a list of equations is not translated into a sequence of nested *"let ... in ..."* declarations, but into simultaneous *"let ... with ..."* declarations specific to *pseudo-ML*.

$\boxed{[\![e]\!]}$

$$\begin{aligned}
&[\![()]\!] \equiv ()\\
&[\![k]\!] \equiv k\\
&[\![x]\!] \equiv x\\
&[\![e \diamond e']\!] \equiv [\![e]\!] \diamond [\![e']\!]\\
&[\![e\ \mathbf{when}\ x]\!] \equiv \mathbf{if}\ x\ \mathbf{then}\ [\![e]\!]\ \mathbf{else}\ \bot\\
&[\![e\ \mathbf{whennot}\ x]\!] \equiv \mathbf{if}\ x\ \mathbf{then}\ \bot\ \mathbf{else}\ [\![e]\!]\\
&[\![\square\, e]\!] \equiv \square\, [\![e]\!]
\end{aligned}$$

$\boxed{[\![ce]\!]}$

$$\begin{aligned}
&[\![\mathbf{if}\ e\ \mathbf{then}\ ce'\ \mathbf{else}\ ce'']\!] \equiv \mathbf{if}\ [\![e]\!]\ \mathbf{then}\ [\![ce']\!]\ \mathbf{else}\ [\![ce'']\!]\\
&[\![\mathbf{merge}\ x\ ce\ ce']\!] \equiv \mathbf{if}\ [\![x]\!]\ \mathbf{then}\ [\![ce]\!]\ \mathbf{else}\ [\![ce']\!]
\end{aligned}$$

$\boxed{[\![ck]\!]}$

$$\begin{aligned}
&[\![\bullet]\!] \equiv \mathbf{true}\\
&[\![ck\ \mathbf{on}\ x]\!] \equiv [\![ck]\!]\ \&\&\ x\\
&[\![ck\ \mathbf{onnot}\ x]\!] \equiv [\![ck]\!]\ \&\&\ (\mathbf{not}\ x)
\end{aligned}$$

$\boxed{[\![\vec{eqn}]\!]_{inits}}$

$$\begin{aligned}
&[\![\varnothing]\!]_{inits} \equiv \varnothing\\
&[\![y \underset{ck}{=} k\ \mathbf{fby}\ e; \vec{eqn}]\!]_{inits} \equiv (y_r = \mathbf{ref}\ k)\ \mathbf{with}\ [\![\vec{eqn}]\!]_{inits}\\
&[\![eqn; \vec{eqn}]\!]_{inits} \equiv [\![\vec{eqn}]\!]_{inits}
\end{aligned}$$

$\boxed{[\![\vec{eqn}]\!]}$

$$\begin{aligned}
&[\![\varnothing]\!] \equiv \varnothing\\
&[\![y \underset{ck}{=} ce; \vec{eqn}]\!] \equiv (y = \mathbf{if}\ [\![ck]\!]\ \mathbf{then}\ [\![ce]\!]\ \mathbf{else}\ \bot)\ \mathbf{with}\ [\![\vec{eqn}]\!]\\
&[\![y \underset{ck}{=} k\ \mathbf{fby}\ e; \vec{eqn}]\!] \equiv (y = \mathbf{if}\ [\![ck]\!]\ \mathbf{then}\ !y_r\ \mathbf{else}\ \bot)\ \mathbf{with}\ [\![\vec{eqn}]\!]
\end{aligned}$$

$\boxed{[\![\vec{eqn}]\!]_{updates}}$

$$\begin{aligned}
&[\![\varnothing]\!]_{updates} \equiv ()\\
&[\![y \underset{ck}{=} k\ \mathbf{fby}\ e; \vec{eqn}]\!]_{updates} \equiv (\mathbf{if}\ [\![ck]\!]\ \mathbf{then}\ (y_r := [\![e]\!])\ \mathbf{else}\ ())\ ;\ [\![\vec{eqn}]\!]_{updates}\\
&[\![eqn; \vec{eqn}]\!]_{updates} \equiv [\![\vec{eqn}]\!]_{updates}
\end{aligned}$$

$\boxed{[\![node]\!]}$

$$[\![\mathbf{node}\ f(\vec{x})\ \mathbf{returns}\ (\vec{y}) = \vec{eqn}]\!] \equiv \mathbf{fun}\ () \rightarrow\ \mathbf{let}\ [\![\vec{eqn}]\!]_{inits}\ \mathbf{in}\ (\mathbf{fun}\ \vec{x} \rightarrow\ \mathbf{let}\ [\![\vec{eqn}]\!]\ \mathbf{in}\ ([\![\vec{eqn}]\!]_{updates}\ ;\ ([\![\vec{y}]\!])))$$

FIGURE 5.4 – Translation Function from OCaLustre to *pseudo-ML*

Typing Correctness in *pseudo-ML*

Based on the functions and rules defined in the previous sections, we now present the various steps needed to prove typing correctness with respect to the translation of an OCaLustre node into a pseudo-ML program.

First, it is proved that any simple expression (e) preserves, under the same environment Γ, the same type after translation :

Lemme 5.1.1 (Typing Correctness of Simple Expressions)**.**

$$\forall\ \mathcal{R}\ \Gamma\ e\ t, (\Gamma \vdash e : t \Leftrightarrow \mathcal{R}, \Gamma \vdash [\![e]\!] : t)$$

From this lemma, we deduce that any conditional expression (ce) also preserves the same type after translation :

Lemme 5.1.2 (Typing Correctness of Conditional Expressions)**.**

$$\forall\ \mathcal{R}\ \Gamma\ ce\ t, (\Gamma \vdash ce : t \Leftrightarrow \mathcal{R}, \Gamma \vdash [\![ce]\!] : t)$$

From these lemmas, we prove that a list of equations preserves the same type after translation, provided that the initialization and update steps of the registers in the generated function (which may appear in the translated equations) are well typed in the environment $\mathcal{R}$. Moreover, the names of the flows defined by these equations must all be distinct : for example, the same flow x cannot be defined twice by two different equations in the same node.

Lemme 5.1.3 (Typing Correctness of Equations)**.** $\forall\ \mathcal{R}\ \Gamma\ \vec{eqn}\ t,$

$$distinct(names(\vec{eqn})) \Rightarrow \mathcal{R}, \varnothing \vdash [\![\vec{eqn}]\!]_{inits} : t \Rightarrow \mathcal{R}, \Gamma \vdash [\![\vec{eqn}]\!]_{updates} : unit \Rightarrow \forall t', (\Gamma \vdash \vec{eqn} : t' \Leftrightarrow \mathcal{R}, \Gamma \vdash [\![\vec{eqn}]\!] : t')$$

We first reason about nodes in which all equations defined in their body correspond to output flows. In other words, we initially deal with nodes that do not have flows whose scope is purely local. A node *node* that does not define locally scoped flows then satisfies the predicate $nolocals(node)$:

$\boxed{nolocals(node)}$

$$\frac{\vec{y} = names(\vec{eqn})}{nolocals(\mathbf{node}\, f(\vec{x})\ \mathbf{returns}\ (\vec{y}) = \vec{eqn})}$$

The combination of the previous lemmas makes it possible to prove the typing correctness lemma for a node, which states that, for any type t, if a node *node* is well formed, then it has type t if and only if its translation has type $unit \rightarrow t$ (in an initially empty environment) :

Lemme 5.1.4 (Typing Correctness of a Node (without locally scoped flows))**.**

$$\forall\ node\ t, \vdash_{wf} node \Rightarrow nolocals(node) \Rightarrow (\vdash node : t \Leftrightarrow \varnothing, \varnothing \vdash [\![node]\!] : unit \rightarrow t)$$

The previous lemma only applies to nodes for which no flow is locally scoped. In other words, the *pseudo-ML* function generated for such nodes returns an n-tuple containing exactly the list of all variables defined in the function. In order to extend this property to any node definition (including those that make use of locally defined flows—and thus are not contained in the list of output variables), we derive from the typing rule for a node presented in Figure 5.3 a new rule that, in particular, allows us to state that the output flows are a subset of the flows computed in a node :

$\boxed{\vdash node : t}$

$$\frac{\vdash \textbf{node}\, f(\vec{x})\, \textbf{returns}\, (\vec{y}) = \vec{eqn} : \vec{t} \rightarrow \vec{t}'' \qquad \vdash_{wf} \textbf{node}\, f(\vec{x})\, \textbf{returns}\, (\vec{y}) = \vec{eqn} \qquad (\vec{z} : \vec{t}') \subseteq (\vec{y} : \vec{t}'')}{\vdash \textbf{node}\, f(\vec{x})\, \textbf{returns}\, (\vec{z}) = \vec{eqn} : \vec{t} \rightarrow \vec{t}'}$$

We can then finally derive the typing correctness lemma for a well-formed OCaLustre node, which may potentially contain locally scoped flows :

Lemme 5.1.5 (Typing Correctness of a Node)**.**

$$\forall\ node\ t, \vdash_{wf} node \Rightarrow (\vdash node : t \Leftrightarrow \emptyset, \emptyset \vdash [\![node]\!] : unit \rightarrow t)$$

We can thus deduce that any OCaLustre program consistent with the typing rules of the language results in a *pseudo-ML* program that is itself well typed.

5.1.2 Translation Using Nested Declarations

The *pseudo-ML* language includes, in addition to the *"let ... with ... in ..."* construction discussed in the previous section, a *"let ... in ..."* construction that only allows the declaration of a single variable, as in the OCaml language. Thus, converting *pseudo-ML* code that uses simultaneous declarations into *pseudo-ML* code that uses a sequence of nested declarations amounts to producing a program whose typing rules are similar to those of an OCaml program. Consequently, proving that the translation of OCaLustre programs into pseudo-ML preserves typing correctness when *"let ... with ... in ..."* is replaced with *"let ... in ..."* makes it possible to verify that the sequential OCaml program generated from an OCaLustre node (detailed earlier in Section 4.4) is itself well typed, since in this case the translation rules into *pseudo-ML* and into OCaml are very close[4].

The function $\ll \cdot \gg$ converting a program containing simultaneous declarations into a program containing nested declarations is described in Figure 5.5.

Nevertheless, the typing equivalence between code containing simultaneous declarations and code containing nested declarations can only be verified provided that there exists an ordering of the nested *"let"* statements such that no variable is used in the declarations of earlier variables before it has been declared. For example, simply replacing *"let ... with ..."* by *"let ... in ..."* in the code snippet presented in the previous section would be incorrect (in both OCaml and pseudo-ML), since it would then refer to the variable y in order to assign a value to the variable x before y itself is defined :

```
let x = y in let y = 2 in x
```

whereas changing the order of the declarations yields correct code with the expected semantics :

4. Even if some differences remain, such as the fact that certain variables are typed in a separate environment.

$\boxed{\ll \delta_r \gg_{e'}}$

$$\begin{array}{l} \ll \emptyset \gg_e \equiv e \\ \ll (y_r = e') \textbf{ with } \delta_r \gg_e \equiv \textbf{let } y_r = e' \textbf{ in } \ll \delta_r \gg_e \end{array}$$

$\boxed{\ll \delta \gg_{e'}}$

$$\begin{array}{l} \ll \emptyset \gg_e \equiv e \\ \ll (y = e') \textbf{ with } \delta \gg_e \equiv \textbf{let } y = e' \textbf{ in } \ll \delta \gg_e \end{array}$$

$\boxed{\ll e' \gg}$

$$\begin{array}{l} \ll \textbf{let } \delta \textbf{ in } e \gg \equiv \ll \delta \gg_{\ll e \gg} \\ \ll \textbf{let } \delta_r \textbf{ in } e \gg \equiv \ll \delta_r \gg_{\ll e \gg} \\ \ll \textbf{fun } \vec{x} \rightarrow e \gg \equiv \textbf{fun } \vec{x} \rightarrow \ll e \gg \\ \ll e \gg \equiv e \end{array}$$

Figure 5.5 – Conversion Function from Simultaneous Declarations to Nested Declarations

```
let y = 2 in let x = y in x
```

The conversion from simultaneous declarations to nested declarations is therefore valid only if there exists an order of appearance of the variables in the simultaneous declarations that prevents any reference to a variable before its definition. This property is in fact exactly equivalent to the notion of "well scheduling" of an OCaLustre program. Indeed, the scheduling step of OCaLustre programs is precisely intended to find a variable declaration order that allows the definition of variables in sequential code.

Simultaneous *pseudo-ML* declarations resulting from the translation of equations of an OCaLustre node can therefore be converted into nested declarations only if these equations are well scheduled :

Theorem 5.1.2 (Equivalence of *let ... with* and *let* (declarations))**.**

$$\forall\, \vec{x}\, \vec{t_x}\, \vec{eqn}, \delta = [\![\vec{eqn}]\!] \Rightarrow \vec{x} \cap vars(\delta) = \emptyset \Rightarrow distinct(vars(\delta)) \Rightarrow \vec{x} \vdash_{ws} eqns \Rightarrow$$
$$\forall\, \Gamma\, t,\ \Gamma = (\vec{x} : \vec{t_x}\,) \Rightarrow (\mathcal{R}, \Gamma \vdash lets\ \delta\ in\ e : t \Leftrightarrow \mathcal{R}, \Gamma \vdash \ll \delta \gg_e : t)$$

Consequently, any *pseudo-ML* program (resulting from the translation of a well-scheduled OCaLustre node) with simultaneous declarations is well typed if and only if an identical program in which the simultaneous declarations are converted into nested declarations is also well typed, and has the same type :

Lemme 5.1.6 (Equivalence of let with and let (node))**.**

$$\forall\, node\ t,\ \vdash_{ws} node \Rightarrow (\vdash [\![node]\!] : t \Leftrightarrow \vdash \ll [\![node]\!] \gg : t)$$

In conclusion, from the lemmas and theorems stated in this section we can deduce that any well-scheduled OCaLustre node is well typed if and only if its translation into an ML language with nested declarations (such as OCaml) is also well typed.

Theorem 5.1.3 (Type Preservation of an OCaLustre Node Translated to OCaml)**.**

$$\forall\ node\ t,\ \vdash_{ws} node \Rightarrow (\vdash node : t \Leftrightarrow \vdash \ll [\![node]\!] \gg : t)$$

This theorem allows us to validate the fact that it is not strictly necessary to implement a type checker specific to OCaLustre, since the OCaml code generated by compiling a node contains all the typing information of the original node. Consequently, any node that is ill-typed in OCaLustre will be detected by the OCaml compiler's type checker.

To extend this type preservation property to complete OCaLustre programs, however, it would be necessary to consider in particular the mechanism of node calls, which are translated into function applications in OCaml.

5.2 Clock Typing Verification

The synchronous clock system, integrated into the OCaLustre language, offers developers the ability to associate *presence conditions* with each flow manipulated by a synchronous program. The clock typing semantics, defined in Section 3.2.4, ensures that operators in the language may only operate (in most cases) on flows driven by the same clock. As a result, adherence to this semantics guarantees that no absent flow value is ever read during program execution, thereby avoiding any erroneous or unpredictable behavior.

In order to assign a clock to each flow in the language, the OCaLustre compiler implements a clock inference algorithm, modeled after those used in Heptagon and SCADE and derived from the work of Colaço and Pouzet [CP03] (based on Milner's $\mathcal{W}$ algorithm [Mil78]). This algorithm statically associates each expression of the language with a synchronous clock, while respecting the clock typing semantics.

To formally ensure that the clocks inferred by the compiler indeed conform to the clocking semantics of the language, we introduce in this section a *clock checker* for OCaLustre. The role of this checker, integrated into the OCaLustre compiler, is to read an abstract syntax tree in which each expression is annotated with its synchronous clock (as inferred by the clock typing algorithm) , and to indicate whether this AST is correct with respect to the semantics of synchronous clock typing. Validation by the clock checker therefore contributes to the safety of the language, by ensuring that the inferred clocks conform to the formal specification given in Section 3.2.4.

The implemented clock checker is integrated into the OCaLustre compilation pipeline immediately after the clock inference process (Figure 5.6) : if the inferred clocks are consistent with the typing rules known to the checker, then the compilation of an OCaLustre program can proceed. Otherwise, the compilation process is halted and the compiler returns an error.

The solution detailed in this section therefore consists of the *a posteriori* verification of the consistency of the clocks assigned to expressions, and thus represents a *certifying* compiler [PSS98] for the safety of inferred clock types. Such a method constitutes an alternative to a *certified* compiler, for which the proof of correctness of the clock inference algorithm implemented in OCaLustre would have been carried out.

It should be noted that verifying the correct clocking of a node is compatible with the possibility for this node to have inputs or outputs that are absent. In this sense, our work is close to recent research that continues the effort of designing a certified Lustre compiler by adding *clocked arguments* [BP19].

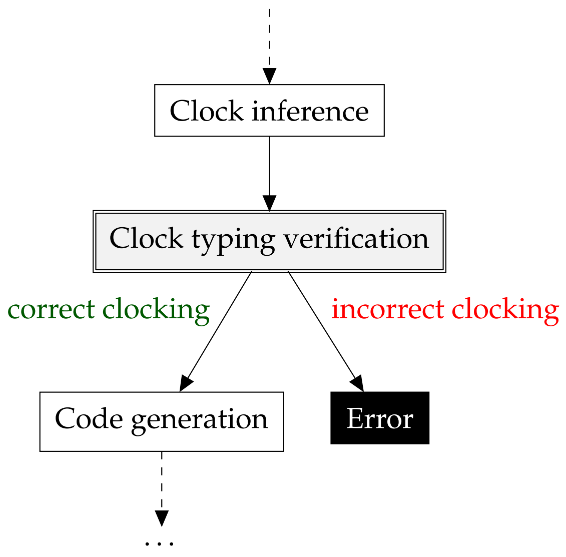


FIGURE 5.6 – Insertion of the clock checker into the compilation chain of an OCaLustre program

5.2.1 Algorithmic rules of correct clocking

The OCaLustre clock checker ensures that the program, represented as an AST annotated with clocks as described in section 4, respects a typing semantics that enforces the correct clocking of flows. This semantics, derived from the typing semantics presented in section 3.2.4, takes into account the annotations added to the OCaLustre program AST during its compilation in order to verify correct clocking.

The *algorithmic* clocking rules for expressions, in this version of the clock-annotated AST, are given in figure 5.7. They are almost identical to those of non-annotated expressions, except for the clocking rules of the *unit* value, a constant, or a type constructor, which take these annotations into account. For example, whereas in its non-annotated version a constant was compatible with any clock type, here a constant k annotated with a clock ck has the clock type ck :

$$\frac{}{\mathcal{C} \vdash k^{ck} : ck}\ \textsc{Const}$$

L'ensemble des règles de cadencement qui concernent les autres catégories syntaxiques est donné dans la figure 5.8. Ces règles de cadencement sont pour la plupart directement dérivées des règles présentées en section 3.2.4, néanmoins la règle qui concerne l'application d'un nœud est plus complexe. Elle doit tenir compte, comme nous l'avons décrit lors de la présentation du système des horloges d'OCaLustre, du mécanisme d'application conditionnelle, ainsi que des diverses substitutions apportées aux noms des variables dans les types d'horloges.

$\boxed{\mathcal{C} \vdash e : ck}$

$$\frac{}{\mathcal{C} \vdash ()^{ck} : ck}\ \textsc{Unit} \qquad \frac{}{\mathcal{C} \vdash k^{ck} : ck}\ \textsc{Const} \qquad \frac{}{\mathcal{C} \vdash X_i^{ck} : ck}\ \textsc{Constr}$$

$$\frac{\mathcal{C}(x) = ck}{\mathcal{C} \vdash x : ck}\ \textsc{Var} \qquad \frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash \Box\, e : ck}\ \textsc{Unop} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e' : ck}{\mathcal{C} \vdash e \diamond e' : ck}\ \textsc{Binop}$$

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\ \mathbf{when}\ x : ck\ \mathbf{on}\ x}\ \textsc{When} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\ \mathbf{whennot}\ x : ck\ \mathbf{onnot}\ x}\ \textsc{Whennot}$$

$\boxed{\mathcal{C} \vdash \vec{e} : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash [e] : ck}\ \textsc{One\_E} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash \vec{e} : ck'}{\mathcal{C} \vdash e, \vec{e} : ck \times ck'}\ \textsc{Cons\_E}$$

$\boxed{\mathcal{C} \vdash_{ce} ce : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash_{ce} e : ck}\ \textsc{Exp} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash_{ce} ce : ck \quad \mathcal{C} \vdash_{ce} ce' : ck}{\mathcal{C} \vdash_{ce} \mathbf{if}\ e\ \mathbf{then}\ ce\ \mathbf{else}\ ce' : ck}\ \textsc{If}$$

$$\frac{\mathcal{C} \vdash x : ck \quad \mathcal{C} \vdash_{ce} ce : ck\ \mathbf{on}\ x \quad \mathcal{C} \vdash_{ce} ce' : ck\ \mathbf{onnot}\ x}{\mathcal{C} \vdash_{ce} \mathbf{merge}\ x\ ce\ ce' : ck}\ \textsc{Merge}$$

Figure 5.7 – Clocking rules, in clock-annotated normal form, for expressions

To represent these substitutions, we introduce the judgment $\vec{x} \underset{\mathcal{S}}{\sim} \vec{e} \rhd \sigma$ which states that, for a set of supports $\mathcal{S}$, a list of formal parameters $\vec{x}$, and a list of actual arguments $\vec{e}$, σ is the substitution of the variable names present in the list of formal parameters of the application with the names of its actual arguments.

The inductive rules related to this judgment are as follows :

$\boxed{\vec{y} \underset{\mathcal{S}}{\sim} \vec{e} \rhd \sigma}$

$$\frac{x \in \mathcal{S}}{x \underset{\mathcal{S}}{\sim} [y] \rhd \{x \mapsto y\}}\ \textsc{Sub\_E\_in} \qquad \frac{x \notin \mathcal{S}}{x \underset{\mathcal{S}}{\sim} [e] \rhd \varnothing}\ \textsc{Sub\_E\_notin}$$

$$\frac{}{() \underset{\mathcal{S}}{\sim} [()^{ck}] \rhd \varnothing}\ \textsc{Sub\_E\_unit} \qquad \frac{y \underset{\mathcal{S}}{\sim} [e] \rhd \sigma \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{e} \rhd \sigma'}{y, \vec{y} \underset{\mathcal{S}}{\sim} e, \vec{e} \rhd \sigma \oplus \sigma'}\ \textsc{Sub\_E\_list}$$

For example, if a node f has the signature $(x : \bullet) \to (y : \bullet \text{ on } x)$, then the application $f\ z$ induces the substitution of x by z for the instantiation of the type of f in this context : consequently, in $f\ z$, f has the type $\bullet \to \bullet$ on z. We can therefore formally derive the judgment $x \underset{\{x\}}{\sim} z \rhd \{x \mapsto z\}$.

Because the output flows of a node may themselves be clocked by other output flows, it is also necessary to substitute the names of the formal output parameters of a node with the actual names on the left-hand side of the equality symbol in the equation corresponding to the node application. The judgment $\vec{y} \underset{\mathcal{S}}{\sim} \vec{y'} \rhd \sigma$ therefore means that, for a set of supports $\mathcal{S}$, σ is the substitution of the formal output parameter names $\vec{y}$ by the actual output flow names $\vec{y'}$ (corresponding to the variable names on the

$\boxed{\mathcal{C} \vdash \vec{y} : ck}$

$$\frac{}{\mathcal{C} \vdash () : \bullet}\ \textsc{NNil} \qquad \frac{\mathcal{C}(y) = ck}{\mathcal{C} \vdash y : ck}\ \textsc{NVariable} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{C} \vdash y, \vec{y} : ck \times ck'}\ \textsc{NList}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash eqn}$

$$\frac{\vec{x} \underset{\mathcal{S}}{\sim} \vec{e} \rhd \sigma_1 \quad ck_1 = \sigma_1(ck'_1[ck]) \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{y}' \rhd \sigma_2 \quad ck_2 = \sigma_2 \oplus \sigma_1(ck'_2[ck])}{ck_1 \rightarrow ck_2 = \underset{(\vec{e}, \vec{y}', ck)}{inst}\left(\forall \bullet.(\vec{x}{:}ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}{:}ck'_2)\right)}\ \textsc{Inst}$$

$$\frac{\mathcal{H}(f) = \forall \bullet .(\vec{x} : ck) \rightarrow (\vec{y} : ck') \quad \mathcal{S} = carriers(ck) +\!\!+\, carriers(ck')}{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck) \xrightarrow{\mathcal{S}} (\vec{y} : ck')}\ \textsc{Sign}$$

$$\frac{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}' : ck'_2) \quad ck_1 \rightarrow ck_2 = \underset{(\vec{e}, \vec{y}', ck)}{inst}\left(\forall \bullet.(\vec{x}{:}ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}{:}ck'_2)\right) \quad \mathcal{C} \vdash \vec{e} : ck_1 \quad \mathcal{C} \vdash \vec{y}' : ck_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y}' \underset{ck}{=} f(\vec{e})}\ \textsc{App}$$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash_{ce} ce : ck}{\mathcal{H}, \mathcal{C} \vdash y \underset{ck}{=} ce}\ \textsc{Expr} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e : ck}{\mathcal{H}, \mathcal{C} \vdash y \underset{ck}{=} k \ \mathbf{fby}\ e}\ \textsc{Fby}$$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e_1 : ck \quad .. \quad \mathcal{C} \vdash e_{n-1} : ck}{\mathcal{H}, \mathcal{C} \vdash y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e\ e_1\ ..\ e_{n-1}}\ \textsc{Call}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash \vec{eqn}}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash eqn}{\mathcal{H}, \mathcal{C} \vdash [eqn]}\ \textsc{OneEqn} \qquad \frac{\mathcal{H}, \mathcal{C} \vdash eqn \quad \mathcal{H}, \mathcal{C} \vdash \vec{eqn}}{\mathcal{H}, \mathcal{C} \vdash eqn;\ \vec{eqn}}\ \textsc{ConsEqns}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash node : \omega}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash \vec{eqn} \quad \mathcal{C} \vdash \vec{x} : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{H}, \mathcal{C} \vdash \mathbf{node} f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn} : \forall \bullet .(\vec{x} : ck) \rightarrow (\vec{y} : ck')}\ \textsc{Node}$$

$\boxed{\mathcal{H} \vdash program}$

$$\frac{}{\mathcal{H} \vdash \varnothing}\ \textsc{EmptyProg} \qquad \frac{\mathcal{H}, \mathcal{C} \vdash \mathbf{node} f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn} : \omega \quad (f : \omega) \cup \mathcal{H} \vdash \vec{nodes}}{\mathcal{H} \vdash (\mathcal{C}, \mathbf{node} f(\vec{x})\ \mathbf{return}\ (\vec{y}) = \vec{eqn});;\vec{nodes}}\ \textsc{ConsProg}$$

FIGURE 5.8 – Clocking rules in the clock-annotated normal form (continued)

left-hand side of the equality in the given equation). The inductive rules governing this judgment apply to a different syntactic category than the previous ones : they handle lists of variable names $\vec{y}$, rather than expressions e. These rules are as follows :

$$\boxed{\vec{y} \underset{\mathcal{S}}{\sim} \vec{y}' \triangleright \sigma}$$

$$\frac{y \in \mathcal{S}}{y \underset{\mathcal{S}}{\sim} y' \triangleright \{y \mapsto y'\}}\ \textsc{Sub\_V\_in} \qquad \frac{y \notin \mathcal{S}}{y \underset{\mathcal{S}}{\sim} y' \triangleright \varnothing}\ \textsc{Sub\_V\_notin}$$

$$\frac{}{() \underset{\mathcal{S}}{\sim} () \triangleright \varnothing}\ \textsc{Sub\_V\_nil} \qquad \frac{y \underset{\mathcal{S}}{\sim} y' \triangleright \sigma \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{y}' \triangleright \sigma'}{y, \vec{y} \underset{\mathcal{S}}{\sim} y', \vec{y}' \triangleright \sigma \oplus \sigma'}\ \textsc{Sub\_V\_list}$$

For example, if the node g has the signature $(x : \bullet) \rightarrow ((y : \bullet) \times (z : \bullet \text{ on } y))$, then in the equation $(a, b) = g(32)$ the type of the instance of g is $\bullet \rightarrow (\bullet \times (\bullet \text{ on } a))$ because a is the effective name used to refer to the first output value of g. This example induces the judgment $(y, z) \underset{\{y\}}{\sim} (a, b) \triangleright \{y \mapsto a\}$.

We also recall that the mechanism of conditional application of a node consists in replacing the base clock in the node's signature by a slower clock in order to slow down its execution. We denote by $\vec{ck}[c]$ the substitution of all occurrences of the base clock ($\bullet$) by the clock c in $\vec{ck}$.

$\boxed{ck[ck']}$

$$\begin{aligned}
&\bullet[ck'] \equiv ck' \\
&(ck\ \mathbf{on}\ x)[ck'] \equiv (ck[ck'])\ \mathbf{on}\ x \\
&(ck\ \mathbf{onnot}\ x)[ck'] \equiv (ck[ck'])\ \mathbf{onnot}\ x \\
&(ck_1 \rightarrow ck_2)[ck'] \equiv (ck_1[ck']) \rightarrow (ck_2[ck']) \\
&(ck_1 \times ck_2)[ck'] \equiv (ck_1[ck']) \times (ck_2[ck'])
\end{aligned}$$

Moreover, we provide the definition of a function *carriers*, which computes the list of supports contained in a clock type :

$\boxed{carriers(ck)}$

$$\begin{aligned}
&carriers(\bullet) \equiv \emptyset \\
&carriers(ck\ \mathbf{on}\ x) \equiv x :: carriers(ck) \\
&carriers(ck\ \mathbf{onnot}\ x) \equiv x :: carriers(ck) \\
&carriers(ck_1 \rightarrow ck_2) \equiv carriers(ck_1) +\!\!+ carriers(ck_2) \\
&carriers(ck_1 \times ck_2) \equiv carriers(ck_1) +\!\!+ carriers(ck_2)
\end{aligned}$$

To make the rule for node application easier to understand, we separate it into three distinct rules :

1. A first rule, Sign

$$\frac{\mathcal{H}(f) = \forall \bullet .(\vec{x} : ck) \rightarrow (\vec{y} : ck') \quad \mathcal{S} = carriers(ck) +\!\!+ carriers(ck')}{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck) \xrightarrow{\mathcal{S}} (\vec{y} : ck')}\ \text{Sign}$$

allows us to retrieve the information necessary for typing the application. It states that, in a global typing environment $\mathcal{H}$, if the function f has the signature $(\vec{x} : ck) \rightarrow (\vec{y} : ck')$, and if the set $\mathcal{S}$ corresponds to the supports in ck and ck', then we can derive the judgment $f : (\vec{x} : ck) \xrightarrow{\mathcal{S}} (\vec{y} : ck')$, which associates the signature of f with its list of supports.

2. A second rule, the instantiation rule Inst :

$$\frac{\vec{x} \underset{\mathcal{S}}{\sim} \vec{e} \rhd \sigma_1 \quad ck_1 = \sigma_1(ck'_1[ck]) \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{y'} \rhd \sigma_2 \quad ck_2 = \sigma_2 \oplus \sigma_1(ck'_2[ck])}{ck_1 \rightarrow ck_2 = \underset{(\vec{e},\vec{y'},ck)}{inst}\left(\forall \bullet .(\vec{x}{:}ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}{:}ck'_2)\right)}\ \text{Inst}$$

states that, for a node with signature $(\vec{x} : ck'_1) \rightarrow (\vec{y} : ck'_2)$ with a set $\mathcal{S}$ of supports, if the following conditions are satisfied :

— σ_1 is the substitution of the formal parameter names $\vec{x}$ of the node by the names of its arguments $\vec{e}$.

— The clock ck_1 corresponds to the result of applying to ck'_1 the substitution σ_1 as well as the substitution of the base clock by the conditional application clock ck of the node.

— σ_2 is the substitution of the formal names $\vec{y}$ of the node's output flows by the effective variable names $\vec{y'}$ actually present on the left-hand side of the equality in the equation that applies the node.

— The clock ck_2 corresponds to the result of applying to ck'_2 the substitutions σ_1 and σ_2, as well as the substitution of the base clock by the conditional application clock ck of the node call.

then the type $ck_1 \rightarrow ck_2$ is a valid *instance* of the type of the node considered.

3. Finally, the third rule, App :

$$\frac{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}' : ck'_2) \quad ck_1 \rightarrow ck_2 = \underset{(\vec{e},\vec{y}',ck)}{inst} \left(\forall \bullet .(\vec{x}:ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}:ck'_2)\right) \quad \mathcal{C} \vdash \vec{e} : ck_1 \quad \mathcal{C} \vdash \vec{y}' : ck_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y}' \underset{ck}{=} f(\vec{e})} \textsc{App}$$

combines the two previous rules : it states, for a node f, that if the type $ck_1 \rightarrow ck_2$ is an instance of the signature of f, and if the parameters of the application are of type ck_1 and the returned values are of type ck_2, then the application $\vec{y}' = f(\vec{e})$, whose conditional application clock is ck, is well-clocked.

As an example, Figure 5.9 shows the derivation tree of the conditional application of the node `count` with parameter expression `1 when c`. In this example, no substitution between the parameter names and the argument names is needed, since the signature of the node `count` contains no support. Nevertheless, in order for the conditional application mechanism to be consistent with clock typing, the base clock $\bullet$ is substituted by the clock of the expression `1 when c` (i.e. ($\bullet$ on c)).

5.2.2 Extraction to OCaml of the clock checker

Once the clock typing rules for the annotated normal form have been defined, the preliminary step in building a type checker relies on extracting the rules defined above into Coq. Using Ott's Coq extraction of the rules allows us to translate them into Coq *inductives*. For example, the following snippet shows the inductive corresponding to the Coq extraction of the clocking rule associated with the application of a binary arithmetic operator.

```
Inductive clk_exp : C → exp → clock → Prop :=
  (...)
  | Binop : forall (C:C) (e:exp) (op:operator) (e':exp) (ck:clock),
    clk_exp C e ck →
    clk_exp C e' ck →
    clk_exp C (Ebinop e op e') ck.
```

From the various rules that govern the well-clockedness of programs, described in the previous section, Ott produces the following Coq inductives :

— *clk_exp* such that *clk_exp* $\mathcal{C}$ *e ck* corresponds to the judgment $\mathcal{C} \vdash e : ck$.
— *clk_cexp* such that *clk_cexp* $\mathcal{C}$ *ce ck* corresponds to the judgment $\mathcal{C} \vdash ce : ck$.
— *Well_clocked_eq* such that *Well_clocked_eq* $\mathcal{H}$ $\mathcal{C}$ *eqn* states that an equation is well-clocked in the global environment $\mathcal{H}$ and the local environment $\mathcal{C}$ (i.e. $\mathcal{H}, \mathcal{C} \vdash eqn$).
— *Well_clocked_node* such that *Well_clocked_node* $\mathcal{H}$ $\mathcal{C}$ *node* states that a node *node* is well-clocked in the global environment $\mathcal{H}$ and the local environment $\mathcal{C}$ (i.e. $\mathcal{H}, \mathcal{C} \vdash node$).

$$\dfrac{\dfrac{incr \notin \varnothing}{incr \underset{\varnothing}{\sim} (1^{\bullet}\ \mathbf{when}\ c) \rhd \varnothing} \qquad \bullet\ \mathbf{on}\ c\ = \bullet[\bullet\ \mathbf{on}\ c] \qquad \dfrac{n \notin \varnothing}{n \underset{\varnothing}{\sim} x \rhd \varnothing} \qquad \bullet\ \mathbf{on}\ c\ = \bullet[\bullet\ \mathbf{on}\ c]}{(\bullet\ \mathbf{on}\ c) \rightarrow (\bullet\ \mathbf{on}\ c) = \underset{(1^{\bullet}\ \mathbf{when}\ c, x, \bullet\ \mathbf{on}\ c)}{inst}((incr : \bullet) \xrightarrow{\varnothing} (n : \bullet))}\ \textsc{Inst}$$

$$\dfrac{\dfrac{\mathcal{H} \vdash count : (incr : \bullet) \rightarrow (n : \bullet) \qquad \varnothing = carriers(\bullet) +\!\!+\ carriers(\bullet)}{\mathcal{H} \vdash count : (incr : \bullet) \xrightarrow{\varnothing} (n : \bullet)}\ \textsc{Sign} \qquad (\ \textsc{Inst}\ ) \qquad \dfrac{\dfrac{}{\mathcal{C} \vdash 1^{\bullet} : \bullet}\ \textsc{Const} \quad \dfrac{(x, \bullet\ \mathbf{on}\ c) \in \mathcal{C}}{\mathcal{C} \vdash c : \bullet}\ \textsc{Var}}{\mathcal{C} \vdash (1^{\bullet}\ \mathbf{when}\ c) : \bullet\ \mathbf{on}\ c}\ \textsc{When} \qquad \dfrac{(x, \bullet\ \mathbf{on}\ c) \in \mathcal{C}}{\mathcal{C} \vdash x : \bullet\ \mathbf{on}\ c}\ \textsc{Var}}{\mathcal{H}, \mathcal{C} \vdash x \underset{\bullet\ \mathbf{on}\ c}{=} count(1^{\bullet}\ \mathbf{when}\ c)}\ \textsc{App}$$

FIGURE 5.9 – Example of derivation of the INST and APP rules for conditional application

However, the generation of inductive relations is not sufficient for our use case : we aim to implement a type checker that must therefore be *executable*, so it can be integrated as a software stage in the compilation sequence of an OCaLustre program. To obtain a computational and executable version of the various clock typing rules, functional versions of these rules have also been implemented in Coq.

For example, the following excerpt of the recursive function `clockof_exp` is the Coq code corresponding to the computation of the clock for the application of a binary operator :

```
Fixpoint clockof_exp (C : clockenv) (e:exp) :=
  match e with
   (...)
   | Ebinop e1 op e2 ⇒
     let c1 := clockof_exp C e1 in
     let c2 := clockof_exp C e2 in
     match c1,c2 with
     | Some a, Some b ⇒ if clock_eqb a b then Some a else None
     end
   end.
```

It is therefore fundamental, in order to guarantee the correctness of our approach, that these executable functions respect the typing semantics formally defined by the inductive rules given in this section. The certification that these rules are respected relies on a proof, in Coq, of the equivalence between the inductive versions and the functional versions of these rules. We have thus proved, in the Coq proof assistant, the following equivalences :

— At the level of expressions : if, in a local environment $\mathcal{C}$, an expression e is associated with the clock ck in the inductive relation clk_exp, then the function $clockof\_exp$, applied to e, also returns the clock ck :

$$\forall\ \mathcal{C}\ e\ ck,\ \mathrm{clockof\_exp}\ \mathcal{C}\ e = Some\ ck \Leftrightarrow clk\_exp\ \mathcal{C}\ e\ ck$$

— This equivalence allows us to deduce that, at the level of control expressions, if, in a local environment $\mathcal{C}$, a control expression ce is associated with the clock ck in the inductive relation clk_exp, then the function $clockof\_cexp$, applied to ce, also returns the clock ck :

$$\forall\ \mathcal{C}\ e\ ck,\ \mathrm{clockof\_cexp}\ \mathcal{C}\ e = Some\ ck \Leftrightarrow clk\_cexp\ \mathcal{C}\ e\ ck$$

— From these relations, we can deduce that if, for a global clock typing environment $\mathcal{H}$ and a local environment $\mathcal{C}$, an equation eqn is well-clocked in the inductive version of the rules, then it is also well-clocked in the functional version :

$$\forall\ \mathcal{H}\ \mathcal{C}\ eqn,\ \mathrm{well\_clocked\_eq}\ \mathcal{H}\ \mathcal{C}\ eqn \Leftrightarrow Well\_clocked\_eq\ \mathcal{H}\ \mathcal{C}\ eqn$$

— We can finally deduce that if a node *node* is well-clocked in the inductive version of the rules, then it is also well-clocked in their functional version :

$$\forall\ \mathcal{H}\ \mathcal{C}\ node,\ \text{well\_clocked\_node}\ \mathcal{H}\ \mathcal{C}\ node \Leftrightarrow Well\_clocked\_node\ \mathcal{H}\ \mathcal{C}\ node$$

The Coq sources representing all the definitions and proofs leading to this result are available online [⚓1].

These properties, once formally proven, provide us with the assurance that our manually defined executable functions respect the clocking semantics represented by the inductive rules originally passed to Ott. Thus, any extraction of these functions into executable code will also preserve this same semantics.

The Coq functions are then extracted into standard OCaml functions. For example, the previous excerpt that handled the clocking of a binary operator is converted into the following piece of OCaml code :

```
let rec clockof_exp c = function
(...)
| Ebinop (e1, _, e2) ->
  let c1 = clockof_exp c e1 in
  let c2 = clockof_exp c e2 in
  (match c1 with
   | Some a ->
     (match c2 with
      | Some b -> if clock_eqb a b then Some a else None
      | None -> None)
   | None -> None)
```

The code extracted from Coq is subsequently linked with some glue code capable of converting certain incompatible types produced by this extraction mechanism (for instance, Coq extracts strings into character lists, which do not correspond to the base `string` type of the OCaml language). Finally, the clock checker code is inserted into the compilation chain, and it is checked, for each node definition, that it respects the clocking semantics of OCaLustre programs by applying the function `well_clocked_node`. If this function returns the value **`false`**, the compiler then generates an error and the program compilation is halted. The clock checker is enabled by the `-check_clocks` option of the OCaLustre compiler. For example, checking the node `merger` (which simply applies its three parameters c, a, and b to the **`merge`** operator) produces the following output :

```
$ ocamlc -ppx "ocalustre -check_clocks" tests/merger.ml
merger :: (c:base * a:(base on c) * b:(base onnot c)) -> (d:base)
Checking of merger : OK
```

The clock checker thus ensures that the clock type checker implemented in OCaLustre, which infers the clocks of each equation, deduces for each of them a clock consistent with the synchronous clock typing system. This checking provides an alternative to implementing a fully verified type checker : if the OCaLustre clock type checker operates correctly, then the clock checker will always compute the value *true*. In cases where the clock checker returns the value *false*, this property can no longer be guaranteed. For all the examples presented in this manuscript, as well as for the tests carried out during our work, the clock checker confirms that the inference is correct. Ultimately, this checking could ensure that no absent value is erroneously manipulated by the program during its execution. However, this property would require connecting the operational semantics of the language with its static clock typing semantics, as discussed in Section 3.2.5.

Chapter Conclusion

The properties verified in this section provide guarantees about the programs produced. These guarantees mainly concern the typing of the data manipulated by an OCaLustre program, whether in terms of the standard typing of values or the clocking of the program's various synchronous components. Thanks to the Coq proofs of these properties, we can ensure that the typing of an OCaLustre program provides the safety expected from the language specification. Such guarantees are particularly valuable in a critical embedded context where user safety is at stake. Furthermore, additional static analyses, which benefit from OMicroB's portable approach, can help ensure the correct behavior of an embedded program. In the next chapter, we present an analysis that makes it possible to bound the execution time of an OCaLustre program. As in the approach adopted in this chapter, we will verify the method used for this analysis with the help of Coq.

6 Worst-Case Execution Time Calculation of an OCaLustre Program

The virtual machine approach adopted in our work allows us to *factorize* several static analyses that can be performed on programs independently of the targeted hardware architectures. Indeed, because executable programs are represented in the form of bytecode, common to all implementations of the virtual machine, these analyses can be carried out directly on the generated bytecode files, thereby abstracting away from the hardware on which the program runs. In this chapter, we illustrate such an analysis, which makes it possible to estimate the Worst-Case Execution Time (*WCET*) of an OCaLustre program. The method used takes advantage of the very simple memory model of microcontrollers and relies on the *compositionality* of the execution time of the program's bytecode : analyzing the distinct costs of each program instruction allows us to deduce its total cost. This analysis, which can easily be adapted to the execution of a program on microcontrollers of different models or architectures, thus has the advantage of being compatible with the portability of the virtual machine approach.

In the following, we present and formalize on an idealized bytecode the method used to measure the worst-case execution time of an OCaLustre program, and we prove its correctness with the help of Coq. This method is then applied to the actual OCaml bytecode instruction language using a software tool named *Bytecrawler*, whose functioning we describe. Finally, we discuss the limitations of the compatibility of WCET analysis with the compilation model previously described, and we present a *dedicated mode* of OCaml code generation that makes it possible, without changing its semantics, to estimate the maximum execution time of any OCaLustre program.

6.1 Formal Validation of the Method in Coq

In this section, we describe and formalize the method we adopt to perform the calculation of the worst-case execution time of a bytecode program resulting from the compilation of an OCaLustre program. For the sake of simplification, both the formalization and the correctness proof of the method are carried out on an *idealized* bytecode language, reduced to a few imperative instructions. We conjecture that the results obtained on this idealized bytecode language can be transposed to the concrete subset of OCaml bytecode instructions generated by the compilation of an OCaLustre program.

6.1.1 Definition of an Idealized Bytecode Language

The idealized bytecode language contains the following seven instructions :

$$instr ::= \mathsf{Init}\ x\ v \mid \mathsf{Assign}\ x\ y \mid \mathsf{Add}\ x\ y \mid \mathsf{Sub}\ x\ y \mid \mathsf{Branch}\ v \mid \mathsf{Branchif}\ x\ v \mid \mathsf{Stop}$$

— The Init instruction initializes a variable with a value, for example $x = 3$.
— The Assign instruction updates the value of a variable with the value of another variable ($x = y$)[1].
— The Add instruction increments the value of a variable x by the value of a variable y (i.e. x += y).
— The Sub instruction decrements the value of a variable x by the value of a variable y (i.e. x -= y).
— The Branch instruction performs a jump in the code.
— The Branchif instruction performs a jump provided that a given variable is different from 0.
— The Stop instruction terminates the execution of the program.

The values manipulated by the language are only integers :

$$values, v, w ::= int\_literal \mid v + w \mid v - w$$

A program state σ corresponds to a triplet containing the program code P (a structure that associates each address with the corresponding instruction of the language), a program counter pc, and a memory M (a structure associating variables with integer values) :

$$\sigma \quad ::= \ (P, pc, M)$$

The small-step operational semantics rules of this language are defined in Figure 6.1.

Execution Traces

In the following, we consider the sequence of all states encountered during the execution of a program written in this language. We call this sequence an *execution trace*.

Definition 6.1.1 (Execution Trace). *An execution trace $\mathcal{T}$ of a program is a sequence of states where each state is the image of the previous one by a transition in the operational semantics of the idealized language.*

Our method applies only to *finite* execution traces of a program. Indeed, it would not be possible to analyze programs whose execution is infinite, since by definition their execution time would not be bounded.

Definition 6.1.2 (Finite Execution Trace). *An execution trace $\mathcal{T}$ is finite (denoted $finite(\mathcal{T})$) if it ends with the* Stop *instruction.*

During program execution, all variable values are known, and the execution trace is *unique*. Moreover, when execution terminates correctly, the last instruction of the associated trace is the Stop instruction. Such a trace is called a *deterministic execution trace*.

Definition 6.1.3 (Deterministic Execution Trace). *The function run, applied to a state σ, returns the deterministic execution trace whose initial state is σ :*

1. Access to a variable is performed in constant time.

$\boxed{\sigma \longrightarrow \sigma'}$

$$\frac{P[pc] = \mathsf{Init}\ x\ v}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v])}$$

$$\frac{P[pc] = \mathsf{Assign}\ x\ y \qquad M[y] = v}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v])}$$

$$\frac{P[pc] = \mathsf{Add}\ x\ y \qquad M[x] = v \qquad M[y] = w}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v{+}w])}$$

$$\frac{P[pc] = \mathsf{Sub}\ x\ y \qquad M[x] = v \qquad M[y] = w}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v{-}w])}$$

$$\frac{P[pc] = \mathsf{Branch}\ v}{(P, pc, M) \longrightarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \qquad M[x] = 0}{(P, pc, M) \longrightarrow (P, pc+1, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \qquad M[x] \neq 0}{(P, pc, M) \longrightarrow (P, v, M)}$$

FIGURE 6.1 – Operational semantics of the idealized language

$\boxed{run(\sigma) = \mathcal{T}}$

$$\frac{P[pc] = \mathsf{Stop}}{run((P, pc, M)) = [(P, pc, M)]} \qquad \frac{\sigma \longrightarrow \sigma' \qquad run(\sigma') = \mathcal{T}}{run(\sigma) = \sigma\ ::\ \mathcal{T}}$$

For example, consider the following program P :

$$[0 : \mathsf{Init}\ \text{"x"}\ 4\ ;\ 1 : \mathsf{Branchif}\ \text{"x"}\ 3\ ;\ 2 : \mathsf{Branch}\ 0\ ;\ 3 : \mathsf{Add}\ \text{"x"}\ \text{"x"};\ 4 : \mathsf{Stop}]$$

The sequence of states $\mathcal{T}$ thus constitutes an execution trace of program P, with the initial state composed of P, a program counter initialized at 0, and an empty memory (i.e. $(P, 0, \varnothing)$) :

$$\mathcal{T} = run((P, 0, \varnothing)) = [(P, 0, \varnothing)\ ;\ (P, 1, [x = 4])\ ;\ (P, 3, [x = 4])\ ;\ (P, 4, [x = 8])]$$

Costs

We associate a *cost* with each instruction of the language. This cost corresponds to the execution time (in number of cycles) of each instruction. Our analysis thus depends on a cost function $cost_{instr}$, which associates an integer value with each instruction :

$$cost_{instr} : instr \rightarrow nat$$

The cost $cost_{step}$ of a transition is equal to the cost of the corresponding instruction :

$$cost_{step}(P, pc, M) = cost_{instr}(P[pc])$$

And the cost *cost* of an execution trace corresponds to the sum of the values returned by the cost function for each instruction in this trace :

$$cost(\mathcal{T}) = \sum_{\sigma \in \mathcal{T}} cost_{step}(\sigma)$$

For example, we represent the cost function of the instructions of the idealized language by the following table, which associates an arbitrary cost with each instruction :

Instruction	Cost (cycles)
Init	4
Assign	2
Add	5
Sub	7
Branch	2
Branchif	3
Stop	1

The cost of the execution trace defined in the previous example, which corresponds to the sum of the instruction costs in that trace, is therefore :

$$\begin{aligned} cost(\mathcal{T}) &= cost_{step}(P, 0, \emptyset) + cost_{step}(P, 1, [x = 4]) + cost_{step}(P, 3, [x = 4]) + cost_{step}(P, 4, [x = 8]); \\ &= cost_{instr}(P[0]) + cost_{instr}(P[1]) + cost_{instr}(P[3]) + cost_{instr}(P[4]) \\ &= cost_{instr}(\textsf{Init "x" 4}) + cost_{instr}(\textsf{Branchif "x" 3}) + cost_{instr}(\textsf{Add "x" "x"}) + cost_{instr}(\textsf{Stop}) \\ &= 13\ cycles \end{aligned}$$

6.1.2 Variable Erasure

In a real, non-trivial program, it is common for the values of certain variables not to be known prior to program execution. Such values may, for example, result from the use of operators with a non-deterministic semantics [CL14], or more commonly, come from the execution environment of the program (user inputs, signals, etc.). In our application context, many values manipulated by the programs indeed originate from the electronic environment of the physical setups that contain a microcontroller : for instance, when the latter reacts to the value of a temperature sensor, this value is unknown at the time of program compilation. It is therefore not always possible to statically and deterministically evaluate the complete execution path of a given program, since values from its environment may condition a branch in the program's control flow : we thus cannot predict which path the program will take at runtime when the value of a condition is not statically known. This non-determinism, introduced by polling the program's environment, makes it difficult to compute the program's total execution time : depending on which path is taken at runtime, the execution duration may vary significantly.

To be able to statically bound the execution time of a program, it is therefore necessary to reason about an abstract representation of this program, by statically representing all possible execution paths. From the resulting set, it is possible to deduce an execution time value that upper-bounds the execution time of all the elements of this set.

The abstract representation of a program then manipulates variables whose values are not known at compilation time. To do this, we extend the grammar of values of the idealized language so as to support *unknown* variable values, which we represent using the symbol $\top$ (*top*) to indicate that they may correspond to *any* value :

$$values, v, w ::= int\_literal \mid v + w \mid v - w \mid \top$$

The program's memory may thus associate some variables with unknown values. We call such a memory an *erasure*, where all or part of the variables are associated with the value $\top$.

Definition 6.1.4 (Erasure). *The following inductive rules define the erasure of a memory :*

$\boxed{M \searrow M'}$

$$\frac{}{\varnothing \searrow \varnothing} \qquad \frac{M \searrow M'}{M[x := v] \searrow M'[x := v]} \qquad \frac{M \searrow M'}{M[x := v] \searrow M'[x := \top]}$$

A memory M' is therefore an erasure of a memory M (denoted $M \searrow M'$) if and only if M and M' share the same set of variables, but some values of the variables known in M are unknown in M'.

In the following, we will use the same notation to represent a state σ' whose memory corresponds to an erasure of the memory contained in a state σ. If the memory of σ' is an erasure of the memory in σ, we then write $\sigma \searrow \sigma'$.

Since the cost of a transition depends only on the program and the program counter, it does not change after erasure of the state's memory :

Lemme 6.1.1. $\forall\, \sigma\ \sigma', \sigma \searrow \sigma' \Rightarrow cost_{step}(\sigma) = cost_{step}(\sigma')$

By extension, the cost of an erased execution trace is identical to the cost of a non-erased trace :

Lemme 6.1.2. $\forall\, \mathcal{T}\ \mathcal{T}',\ \mathcal{T} \searrow \mathcal{T}' \Rightarrow cost(\mathcal{T}) = cost(\mathcal{T}')$

6.1.3 Non-deterministic Evaluation

The appearance of unknown values ($\top$) introduces non-determinism into the semantics of the language. Indeed, if for a conditional branch instruction (Branchif x v), the value of the condition x is erased, then it is no longer possible to know statically whether the program should follow the path corresponding to the case where x is true (i.e. $x \geq 1$), or the one where x is false (i.e. $x = 0$). The total execution cost of a program can therefore, as soon as multiple execution paths are possible, diverge considerably depending on which path is taken. For example, consider the following program :

P = [Init "a" 12 ; 1 : Branchif "x" 5 ; 2 : Init "y" 42 ; 3 : Branch 4 ; 4 : Stop]

If the value of the variable `x` is not known at program compilation (i.e. $M[x] = \top$), then it is not possible to predict its exact execution path at runtime. From the branch conditioned by the value of `x`, two different execution paths are possible :

$$\begin{tikzcd}
 & (P, 0, M) \arrow[d] & \\
 & (P, 1, M[a := 12]) \arrow[dl, "x = 0"'] \arrow[dr, "x \neq 0"] & \\
(P, 5, M[a := 12]) & & (P, 2, M[a := 12]) \arrow[d] \\
 & & (P, 3, M[a := 12][y := 42]) \arrow[d] \\
 & & (P, 4, M[a := 12][y := 42])
\end{tikzcd}$$

The cost of P is therefore potentially different depending on the actual path taken : if we keep the same cost function as in the previous example, then the positive branch (the one that jumps directly to $pc = 5$) has a cost of 8, while the negative branch has a cost of 14.

To make it possible to compute an upper bound of a program's execution cost, we introduce the notion of the *maximum cost* of an execution. This cost is computed by summing the costs of the various states encountered while following the transitions of the operational semantics of the idealized language. In the presence of a branch whose condition is unknown, the maximum cost is calculated by following the transition toward the branch with the higher cost.

Definition 6.1.5 (Maximum Cost). *The function $cost_{max}$ computes the maximum cost of a program, starting from a state σ :*

$\boxed{cost_{max}(\sigma) = c}$

$$\frac{P[pc] = \mathsf{Stop}}{cost_{max}((P, pc, M)) = cost_{instr}(\mathsf{Stop})} \qquad \frac{\sigma \longrightarrow \sigma' \quad cost_{step}(\sigma) = c \quad cost_{max}(\sigma') = k}{cost_{max}(\sigma) = c + k}$$

$$\frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \top \\ cost_{step}((P, pc, M)) = c \quad cost_{max}((P, pc + 1, M)) = k \quad cost_{max}((P, v, M)) = k' \end{array}}{cost_{max}((P, pc, M)) = c + max(k, k')}$$

By abuse of notation, for readability, we use the equality symbol in the previous rules. However, it should be noted that the function $cost_{max}$ is partial, and is defined in Coq as an inductive relation. The definition of this function allows us to state the following theorem, which asserts that the maximum execution cost, computed regardless of the erasure applied to a program's initial memory, constitutes an upper bound of the program's actual execution cost.

Theorem 6.1.1 (Correctness). $\forall\ \sigma\ \mathcal{T}, run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\mathcal{T})$

This theorem forms the basis of the WCET calculation of our programs : if the cost function associates each instruction with its execution time, and if the initial memory associates each variable with the value $\top$, then the corresponding maximum cost is an upper bound of the program's worst-case execution time (since a state whose memory contains only unknown values is an erasure of all possible initial states).

The remainder of this section is devoted to the proof of this theorem, which establishes the correctness of the WCET calculation method.

6.1.4 Proof of Correctness

The Coq sources containing the various formal representations of the notions introduced in this section, as well as the proofs of the associated lemmas and theorems, are available online [⚓1].

Non-deterministic Semantics of the Idealized Language

As illustrated earlier, the introduction of unknown values ($\top$) in the idealized language induces non-determinism in its execution : some erasures may lead to different execution traces which, although all valid, may vary greatly in cost. To represent this non-determinism, we present a *non-deterministic semantics* for the idealized language, capable of handling *unknown* values. This semantics, whose evaluation rules are described in Figure 6.2, corresponds both to the direct transposition of the deterministic rules defined in Figure 6.1[2], as well as the addition of two rules to handle the case where the condition of a conditional branch instruction (Branchif) is unknown. The premises of the conditional branch rule handling the *false* case and those of the rule handling the *true* case are indistinguishable when the value of the branch condition is unknown. The introduction of the notion of values unknown at compilation time, combined

2. With the exception that addition and subtraction operations are extended to return the value $\top$ as soon as one of the operands is unknown : for example, $x \mathbf{+} \top = \top$. To denote this behavior, arithmetic operators will be written in bold.

with the addition of these two rules, thus provides the expected non-deterministic character to the semantics of the instructions of the idealized language.

$\boxed{\sigma \rightsquigarrow \sigma'}$

$$\frac{P[pc] = \textsf{Init}\ x\ v}{(P, pc, M) \rightsquigarrow (P, pc+1, M[x := v])}$$

$$\frac{P[pc] = \textsf{Assign}\ x\ y \qquad M[y] = v}{(P, pc, M) \rightsquigarrow (P, pc+1, M[x := v])}$$

$$\frac{P[pc] = \textsf{Add}\ x\ y \qquad M[x] = v \qquad M[y] = w}{(P, pc, M) \rightsquigarrow (P, pc+1, M[x := v + w])}$$

$$\frac{P[pc] = \textsf{Sub}\ x\ y \qquad M[x] = v \qquad M[y] = w}{(P, pc, M) \rightsquigarrow (P, pc+1, M[x := v - w])}$$

$$\frac{P[pc] = \textsf{Branch}\ v}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

$$\frac{P[pc] = \textsf{Branchif}\ x\ v \qquad M[x] = 0}{(P, pc, M) \rightsquigarrow (P, pc+1, M)}$$

$$\frac{P[pc] = \textsf{Branchif}\ x\ v \qquad M[x] \neq 0}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

$$\frac{P[pc] = \textsf{Branchif}\ x\ v \qquad M[x] = \top}{(P, pc, M) \rightsquigarrow (P, pc+1, M)}$$

$$\frac{P[pc] = \textsf{Branchif}\ x\ v \qquad M[x] = \top}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

FIGURE 6.2 – Non-deterministic semantics of the idealized language

Preservation of Transitions and Traces

Our reasoning concerns the maximum cost of the execution traces of a program whose memory contains erased variables : values coming from the environment, unknown prior to program execution. However, during the actual execution of the program, the memory contains no unknown values, and the corresponding execution trace is the unique sequence of transitions in the deterministic operational semantics of the idealized language. It is therefore important, in order for reasoning applied to the program's *non-deterministic* traces to be meaningful, to ensure that the actual execution trace of the program is indeed contained within the set of considered non-deterministic traces. In other words, reasoning on *erased* versions of the memory can only be useful if, among all valid erased traces computed, one of them corresponds to the actual trace of the program (otherwise, nothing could be concluded regarding the program's real execution).

We begin by reasoning at the level of the language semantics transitions. To this end, we first define a lemma stating that a transition in the deterministic semantics persists after being embedded into the non-deterministic semantics of the idealized language. This property is straightforward, since the non-deterministic semantics corresponds to an extension of the deterministic semantics of the idealized language :

Lemme 6.1.3. $\forall\, \sigma\, \sigma',\ (\sigma \longrightarrow \sigma') \Rightarrow (\sigma \rightsquigarrow \sigma')$

Moreover, every transition is preserved after erasure of the program's memory : if there exists a transition from a state σ_1 to a state σ_2 in the non-deterministic semantics, then for any erasure σ'_1 of the original state there exists a transition leading to a state σ'_2, and the latter is an erasure of σ_2 :

Lemme 6.1.4. $\forall\, \sigma_1\, \sigma_2\, \sigma'_1, (\sigma_1 \rightsquigarrow \sigma_2) \wedge (\sigma_1 \searrow \sigma'_1) \Rightarrow (\exists\, \sigma'_2,\ (\sigma'_1 \rightsquigarrow \sigma'_2) \wedge (\sigma_2 \searrow \sigma'_2))$

By combining the two previous lemmas, we can then deduce that every transition in the deterministic semantics is preserved after embedding into the non-deterministic semantics, even after partial or complete erasure of the memory of the original state :

Lemme 6.1.5 (Preservation). $\forall\, \sigma_1\, \sigma_2\, \sigma'_1, (\sigma_1 \longrightarrow \sigma_2) \wedge (\sigma_1 \searrow \sigma'_1) \Rightarrow (\exists\, \sigma'_2,\ (\sigma'_1 \rightsquigarrow \sigma'_2) \wedge (\sigma_2 \searrow \sigma'_2))$

This property is illustrated schematically in Figure 6.3.

$$\begin{tikzcd}
\sigma_1 \arrow[rr] \arrow[dr, dashed] & & \sigma_2 \arrow[dr, dashed] & \\
& \sigma'_1 \arrow[rr, squiggly] & & \sigma'_2
\end{tikzcd}$$

FIGURE 6.3 – Transition preservation

Finally, as illustrated in Figure 6.4, the extension of the above properties to execution traces is straightforward.

$$\begin{tikzcd}
\sigma_1 \arrow[rr] \arrow[dr, dashed] & & \sigma_2 \arrow[rr] \arrow[dr, dashed] & & \cdots \arrow[rr] & & \sigma_n \arrow[dr, dashed] & & \\
& \sigma'_1 \arrow[rr, squiggly] \arrow[dr, dashed] & & \sigma'_2 \arrow[rr, squiggly] \arrow[dr, dashed] & & \cdots \arrow[rr, squiggly] & & \sigma'_n \arrow[dr, dashed] & \\
& & \sigma''_1 \arrow[rr, squiggly] & & \sigma''_2 \arrow[rr, squiggly] & & \cdots \arrow[rr, squiggly] & & \sigma''_n
\end{tikzcd}$$

FIGURE 6.4 – Trace preservation

Upper Bound on the Cost of the Program's Actual Execution

To verify that the function $cost_{max}$ indeed refers to the most costly execution trace of the program, we formally define such a trace, called the *maximal execution trace*.

Definition 6.1.6 (Maximal Trace). *The function run_{max}, which computes the maximal execution trace, is defined inductively as follows :*

$$\boxed{run_{max}\,(\sigma) = \mathcal{T}}$$

$$\frac{\sigma \longrightarrow \sigma' \qquad run_{max}\,(\sigma') = \mathcal{T}}{run_{max}\,(\sigma) = (\sigma \;::\; \mathcal{T})} \qquad \frac{P[pc] = \textsf{Stop}}{run_{max}\,((P, pc, M)) = [(P, pc, M)]}$$

$$\frac{\begin{array}{c} P[pc] = \textsf{Branchif}\ x\ v \qquad M[x] = \top \\ run_{max}\,((P, pc+1, M)) = \mathcal{T} \qquad run_{max}\,((P, v, M)) = \mathcal{T}' \qquad cost(\mathcal{T}) > cost(\mathcal{T}') \end{array}}{run_{max}\,((P, pc, M)) = ((P, pc, M) \;::\; \mathcal{T})}$$

$$\frac{\begin{array}{c} P[pc] = \textsf{Branchif}\ x\ v \qquad M[x] = \top \\ run_{max}\,((P, pc+1, M)) = \mathcal{T} \qquad run_{max}\,((P, v, M)) = \mathcal{T}' \qquad cost(\mathcal{T}) \leq cost(\mathcal{T}') \end{array}}{run_{max}\,((P, pc, M)) = ((P, pc, M) \;::\; \mathcal{T}')}$$

As with $cost_{max}$, the function run_{max} is a partial function defined inductively in Coq.

The maximal trace contains the same states as those considered by the function $cost_{max}$: it follows the paths for which the total sum of costs is the greatest. Its cost is therefore identical to the maximum execution cost :

Lemme 6.1.6. $\forall\ \sigma,\ cost_{max}(\sigma) = cost(run_{max}(\sigma))$

By definition, the maximal trace is a finite trace. Moreover, its cost is greater than that of any other finite trace starting from the same state :

Lemme 6.1.7. $\forall\ \sigma\ \mathcal{T}\ k,\ finite(\sigma :: \mathcal{T}) \Rightarrow cost_{max}(\sigma) = k \Rightarrow k \geq cost(\sigma :: \mathcal{T}))$

By applying the preservation lemma to execution traces, we then derive the main property of the correctness proof of our method : the cost of the maximal execution trace is greater than the cost of any finite trace starting from the same state, even when considering an erasure of its memory.

Lemme 6.1.8. $\forall\ \sigma\ \mathcal{T}\ k, finite(\sigma :: \mathcal{T}) \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') = k \Rightarrow k \geq cost(\sigma :: \mathcal{T})$

Since the deterministic trace is finite, the combination of the previous lemmas finally allows us to conclude with the correctness theorem, which states that the maximal cost of a program, estimated from an erasure of its initial memory, is an upper bound of the program's actual execution cost :

Theorem 6.1.2 (Correctness). $\forall\ \sigma\ \mathcal{T}\ k, run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') = k \Rightarrow k \geq cost(\mathcal{T})$

6.2 Application to the WCET Calculation of an OCaLustre Program : the *Bytecrawler* Tool

The method for calculating worst-case execution time described in the previous section applies to an idealized language limited to a few instructions, but which nonetheless allows, through its branching instructions, a program to loop during its execution. The absence of non-statically bounded loops in the program is, however, an essential condition for ensuring the finiteness of the traces considered : otherwise, some erasures of memory[3] could lead to infinite execution of the static analyzer (which would then be indefinitely computing the execution trace of maximal cost of the program). The analysis described in the previous section was therefore only suitable for *finite* traces, requiring that the program not loop indefinitely. Our analysis is thus applicable only under the condition that we can guarantee that no unbounded loops or recursive calls are possible in the considered programs.

During the compilation of OCaLustre, each node is converted into a non-recursive OCaml function and, although the main program is executed in a main loop where one iteration is performed at each synchronous instant, no nested loops are executed within a single instant : the program simply computes, *instantaneously*, output values from input values. The execution traces of a synchronous instant are therefore *finite*, and under these conditions, it is possible to adapt to OCaLustre the analyses formalized in the previous section, applying them not to the idealized language, but to the instruction set of the OCaml virtual machine, since this is the target (after an initial conversion into standard OCaml code) of the OCaLustre program.

Although more numerous and potentially more powerful, the instructions of the OCaml virtual machine generated by OCaLustre are comparable to those of the idealized language : similarly, some of them perform reference value updates (`SETFIELD`), arithmetic computations (`ADDINT`), or conditional branches (`BRANCHIF`). The analyses carried out on the idealized language are therefore transposable to the instructions of the OCaml virtual machine, and the calculation of the worst-case execution time of a synchronous instant of an OCaLustre program then consists, as in the analysis of the idealized language, of computing the maximal execution cost of the synchronous instant. This cost thus constitutes the worst-case execution time of one instant of the program.

It should be noted that the analysis presented here is valid because the microcontrollers we consider have very simple memory models. Indeed, the absence of cache systems makes the duration of each instruction predictable [BN94], without having to consider scenarios where memory accesses could vary in cost, and ensures the execution times, in number of cycles, of the different instructions of the language. In this sense, the considered targets are said to be free of *timing anomalies* [RWT+06], which enables the compositionality of execution time analyses.

6.2.1 Instruction Cost Calculation

When formalizing the worst-case execution time analysis, we discussed the existence of a specialized cost function, associating an integer value with each instruction of the idealized language. In the OCaLustre execution context, it is therefore necessary to use such a function that associates each instruction of the virtual machine with a maximal execution time, in order to deduce an upper bound on the execution time of the complete program. To this end, we leverage existing static analyzers capable of assigning a maximum number of cycles to a compiled program. For illustration, we used the software tool *Bound-T* [HS02], a memory and execution-time analyzer capable of computing the WCET of a program compiled for an AVR microcontroller.

Our approach consisted of having Bound-T compute the worst-case execution time of each code block responsible for interpreting an instruction of the OMicroB virtual machine. In this way, Bound-T allows

3. For example, the erasure of the variable conditioning the execution of a loop iteration.

us to associate with each virtual machine instruction a value corresponding to the maximum number of machine cycles required for its execution. It should be noted that the execution time of certain OCaml virtual machine instructions depends partly on the value of one of their parameters (for example, the instruction `APPTERM n v` executes `n` iterations of a loop in the interpreter's associated code). One way of handling such instructions is to calculate their cost when their parameter is set to the largest possible value. However, this method results in an upper bound potentially far from reality. In order to assign costs to each instruction more precisely, we recompute the costs of such instructions multiple times by providing different fixed values as their parameters. This method is slower to perform, but it only needs to be done once : once Bound-T has been executed for each virtual machine instruction, the tool is no longer needed, since each OCaml bytecode program will contain only a subset of this instruction set whose costs we have precomputed.

6.2.2 *Bytecrawler* : an Abstract Bytecode Interpreter

Once the Bound-T analysis is complete, we have at our disposal a complete table representing the virtual machine instructions and their cost (in number of cycles). An excerpt of this table, generated for the AVR ATmega2560 microcontroller for a 16-bit version of OMicroB, is reproduced in Figure 6.5. This table constitutes the cost function described in the previous section. It is then possible to replicate the maximal cost trace calculation method, considering that absent values ($\top$) correspond to the results of calls to C primitives (via `CCALL` instructions) used to communicate with the environment.

We therefore propose a static analysis tool, named *Bytecrawler*, which executes the program in the same manner as the function $cost_{max}$ described abovefor example, by exploring both branches of a conditional in order to deduce the one with the maximum cost. *Bytecrawler* thus makes it possible to compute, by accumulating the values provided by the cost function calculated by *Bound-T*, the cost (in number of cycles) of the maximal execution trace of an OCaLustre instant. By extending the result of the correctness theorem to the instruction language of the OCaml virtual machine, we can deduce that the cost thus obtained corresponds to an estimate of the worst-case execution time of a synchronous instant of the OCaLustre program.

The `wcet` function in Figure 6.6 is a simplified excerpt of the function at the core of *Bytecrawler* : it evaluates a program by following the standard semantics of OCaml bytecode instructions, while also being capable of handling *unknown* variable values, resulting from calls to C input/output primitives.

In addition to associating a number of cycles with each instruction of the OCaml virtual machine, it is also necessary to calculate the cost of all the input/output primitives integrated into the virtual machine's standard library. These primitives, written in C, are generally compatible with analyzers such as *Bound-T*. Just as the instruction set does not change over time, the input/output primitivesbeing very low level (often consisting simply of writing or reading values on the microcontroller pins)are not expected to change from one program to another. The tool *Bytecrawler* must therefore, whenever it encounters a `CCALL` instruction invoking a primitive, consult a second table that associates each C primitive name with its estimated execution time. A developer wishing to define custom C primitives must then provide their associated costs in this table.

The approach used by *Bytecrawler* offers a significant advantage over traditional WCET tools designed for programs compiled into native code : since the costs associated with each virtual machine instruction are fixed, they can be reused to analyze the cost of other programs, without forcing the programmer

Bytecode instruction	Cost (in cycles)
ACC	46
PUSH	43
PUSHACC	74
POP	42
ASSIGN	52
ENVACC	50
PUSHENVACC	78
APPLY	50
RETURN	85
GETFIELD	50
SETFIELD	67
GETVECTITEM	54
SETVECTITEM	69
BRANCHIF_1B	36
BRANCHIF_2B	46
BRANCHIF_4B	67
CONST0	25
ADDINT	51
SUBINT	51
MULINT	59

FIGURE 6.5 – Instruction bytecode cost table computed by Bound-T (excerpt)

```
let rec wcet state =
  let state' = {state with pc = state.pc + 1} in
  let instr = state.instrs.(pc) in
  cost instr + match instr with
  | CONST i -> wcet {state' with accu = Int i}
  | BRANCH ptr -> wcet {state with pc = ptr}
  | BRANCHIF ptr ->
    (match state.accu with
    | Int 0 -> wcet state'
    | Int _ -> wcet {state with pc = ptr}
    | Unknown -> max (wcet {state with pc = ptr}) (wcet state'))
  | ADDINT ->
    (match state.accu, state.stack with
    | Int x, (Int y)::s -> wcet {state' with accu = Int (x+y); stack = s}
    | _, _::s -> wcet {state' with accu = Unknown ; stack = s})
  | C_CALL1 _ -> wcet {state' with accu = Unknown}
  | STOP -> 0
```

FIGURE 6.6 – Excerpt of the WCET calculation function

to rerun the sometimes complex machinery of an analyzer such as *Bound-T* after every change to the program. Moreover, the WCET of a program can be computed for several different microcontroller models, as long as *Bytecrawler* is provided with a bytecode instruction cost table adapted to each of these models. Such factorization of analyses is particularly valuable in the context of development for embedded systems, where the targets (circuit, type of microcontroller, . . .) may evolve as needs change.

6.2.3 Dedicated Mode of the OCaLustre Compiler

Since the virtual machine used relies on a *garbage collector*, the calculation of these costs could be inaccurate. Indeed, when allocating a new value in the heap, the memory reclamation algorithm can potentially be triggered, and several machine cycles may be consumed until it completes its execution. If not taken into account, this unpredictable triggering of the garbage collector can lead to errors in the WCET calculation of a program. Several methods can be employed to account for the existence of such an automatic memory management mechanism : one of them consists in simply assuming that a full memory collection is executed at each heap allocation. This over-approximation makes it possible to correctly compute an upper bound of the execution time of a synchronous instant, but it lacks precision, as it is unrealistic to assume that the *garbage collector* triggers so frequently.

It is interesting to note that, in the context of OCaLustre compilation, the values requiring allocation in the heap (typically, registers resulting from the use of the operator ⋙) are declared during the initialization phase of the node : these values represent the environment of the closure generated by the compilation of a node. For example, in the following node, only the register value used to store the previous value of the expression `c + 1` needs to be preserved in memory :

```
let%node count () ~return:c =
  c = (0 ⋙ (c + 1))
```

And this characteristic becomes apparent after compilation : the variable `c_fby` is the only one present in the environment of the closure returned by the function `count` :

```
let count () =
  let c_fby = ref 0 in
    fun () ->
      let c = !c_fby in
      c_fby := c + 1;
      c
```

As a result, it is possible to consider that, in the case where only base-type values are used (such as integers, floats[4], or booleans), the number of values to be allocated in memory for the execution of a node is known prior to program execution. Thus, if the necessary registers are allocated before the execution of the OCaLustre program, no new allocation will be performed during the execution of the instant. This would make it possible to claim that it is not necessary to account for the time taken by the garbage

4. Thanks to our representation of values in OMicroB, floats do not, in fact, need to be allocated on the heap

collector in the calculation of the worst-case execution time of the synchronous instant, since no heap allocation is performed in that instant.

Nevertheless, the compilation method described in Section 4.4 is not entirely compatible with this statement : the compiled code of a node may indeed contain tuples corresponding to its parameters as well as its output flows. These tuples are therefore allocated in the heap at each instant, and the question of triggering the garbage-collection algorithm arises once again whenever a node returns and/or receives more than one flow.

For example, consider the following node :

```
let%node triple (x) ~return:(a,b,c) =
  a = x;
  b = x + 1;
  c = 42
```

This node, when compiled, generates an OCaml function that produces a closure, which in turn induces the creation of a triple `(a,b,c)` whose memory representation leads to a heap allocation at each synchronous instant :

```
let triple () =
  fun x ->
    let a = x in
    let b = x + 1 in
    let c = 42 in
    (a,b,c)
```

Thus, the analysis presented in this chapter is only compatible with the compilation model described in Section 4.4 under the condition that the considered nodes do not have multiple input or output parameters. This strong limitation seems too restrictive to make *Bytecrawler* truly usable on non-trivial programs. Consequently, in order to avoid such allocations occurring during an instant, we present an alternative compilation scheme for OCaLustre nodes. This new compilation method preserves the semantics of the OCaLustre language, but this time allows us to guarantee that no memory allocation is performed during a synchronous instant, regardless of the shape of the nodes.

In this alternative compilation scheme, each node *n* leads to the definition of an OCaml type *n*`_state` corresponding to the state of an instance of that node. This state contains mutable registers corresponding to the node's outputs and to the internal registers necessary for its execution (such as those resulting from the use of the *followed-by* operator) :

```
 type ('a, 'b, 'c) triple_state = {
  mutable triple_out_a: 'a ;
  mutable triple_out_b: 'b ;
  mutable triple_out_c: 'c }
```

The code of the node is then split into two distinct functions. The first is an initialization function, which corresponds to the allocation of the node's state. Since at this stage of compilation the types of the values have not yet been inferred, the value `Obj.magic ()` is used again, except for constant flows (initialized with their value), or equations of the form `k ⋙ e` (which are initialized with the constant `k`). The role of `Obj.magic ()` here is only to reserve on the heap the space needed to store the mutable state of the node : it will never be evaluated during the execution of the synchronous program.

```
let triple_alloc x = {
    triple_out_a = Obj.magic ();
    triple_out_b = Obj.magic ();
    triple_out_c = 42 }
```

This function generates the initial state of the node's registers. Following a state-passing style of execution, the internal state of the program is then a parameter of the node's *step* function, which computes at each instant the values of the flows defined by the node :

```
let triple_step state x =
  let a = x in
  let b = x + 1 in
  let c = 42 in
  state.triple_out_a ← a;
  state.triple_out_b ← b;
  state.triple_out_c ← c
```

This function is responsible, at each instant, for updating (through side effects) the values of the different variables present in the node's state. Since the node's state is a record type whose fields are mutable, each of these fields is allocated when the state is generated, and no new memory allocation occurs in subsequent instants. Thanks to this execution model, which generates code with more *imperative* aspects, the WCET calculation of an instant can thus be applied to the generated *n*`_step` function.

The code of the main function of the generated OCaml program, compatible with this compilation mode, has the following form :

```
let () =
  (* generation of the main node's state *)
  let _st = triple_alloc () in
  while true do
   (* reading inputs *)
   let x = input_triple_x () in
   (* executing the node *)
   triple_step _st x;
   (* writing outputs *)
   output_triple _st.triple_out_a _st.triple_out_b _st.triple_out_c
 done
```

The -na option of the OCaLustre compiler enables the *non-allocating* compilation mode described in this section.

6.2.4 Illustrative Example

In this section, we illustrate the operation of *Bytecrawler* on a short example. The OCaLustre program of this example is reduced to a single node named `count`, which represents a counter : at each synchronous instant, it computes the successor of the integer computed at the previous instant. In addition, this counter can be reset to zero whenever the `reset` parameter of the node is true.

The code of this node (on the left), as well as the set of bytecode instructions into which this node is translated following the compilation of OCaLustre into an OCaml program and then the standard compilation of the latter (on the right[5]), are presented below :

```
let%node count reset ~return:c =
  c = (0 ⋙ if reset then 0 else (c + 1))
```

```
L1:  GRAB 1
     ACC 0
     GETFIELD 0
     PUSH
     ACC 2
     BRANCHIFNOT L7
     CONST 0
     BRANCH L6
L7:  ACC 0
     OFFSETINT 1
L6:  PUSH
     ACC 1
     PUSH
     ACC 3
     SETFIELD 1
     ACC 0
     PUSH
     ACC 3
     SETFIELD 0
     CONST 0
     RETURN 4
```

We now declare two OCaml functions responsible for handling the inputs and processing the outputs of this synchronous program. The first defines the input value of the node (`reset`) as the result of the test that checks whether pin number 0 of port *B* is poweredfor example, if a push button connected to this pin has been pressed :

```
(* input function *)
let input_count_reset () =
   read_bit PORTB PB0
```

```
L4:  CONST0
     PUSH
     CONST1
     CCALL caml_avr_read_bit, 2
     RETURN1
```

Since the value returned by reading this pin (via the call to the primitive `caml_avr_read_bit`) cannot be known at compilation time, it will be represented in *Bytecrawler* as an unknown value ($\top$).

5. The *-dinstr* option of the standard compiler allows the display of this *labeled* bytecode.

For the sake of simplicity, we declare that the second function, which is supposed to handle the value c computed by the synchronous instant, does nothing, that is, it simply computes the value `unit` :

```
(* fonction de sortie *)
let output_count c = ()
```

```
L3: CONST 0
    RETURN 1
```

After initializing the state of the synchronous program, according to the compilation model described in the previous section, the OCaml program repeatedly reads the program input, calls the step function of the synchronous instant, and calls the function handling its outputs. In order for *Bytecrawler* to recognize in the bytecode where each synchronous instant begins and ends, the program loop is replaced, for the purpose of WCET analysis, by calls to the primitives `begin_loop` and `end_loop` :

```
let () =
  let _st = count_alloc () in
  begin_loop ();
    let reset = input_count_reset () in
    count_step _st reset;
    output_count _st.count_out_cpt;
  end_loop ()
```

The bytecode associated with the code enclosed by the calls to `begin_loop` and `end_loop` is then as follows :

```
CCALL begin_loop, 1
CONST 0
PUSH
ACC 5
APPLY 1
PUSH
ACC 0
PUSH
ACC 2
PUSH
ACC 4
APPLY 2
ACC 1
GETFIELD 1
PUSH
ACC 5
APPLY 1
CONST 0
CCALL end_loop, 1
```

Bytecrawler executes the program from the very first bytecode instruction, but begins counting the costs of the instructions encountered only from the call to the `begin_loop` primitive, and continues until the call to `end_loop`, thereby making it possible to compute the cost (in cycles) of the synchronous instant.

Thus, with an instruction cost table computed by *Bound-T* for an ATmega2560, the maximal number of cycles estimated by *Bytecrawler* is 3080. Since such a microcontroller has a clock frequency of 16 MHz, the worst-case execution time of *one* synchronous instant of this program is therefore 1.92×10^{-4} seconds (192 μs). Hence, if the interval between two changes of the state of pin *PB0* is greater than 192 microseconds, the synchronous hypothesis holds.

Chapter Conclusion

In this chapter, we leveraged the factorization of analyses provided by representing programs in bytecode form to compute the worst-case execution time of a program. The proof of correctness of the method ensures its viability, and a software tool following this method has been implemented. Of course, one of the processes required to certify the operation of this tool would be to formalize and prove the method not only on the idealized bytecode presented in this section, but on the actual subset of the OCaml bytecode instruction language produced by the compilation of OCaLustre. Although this would be a significant undertaking due to the number of OCaml bytecode instructions, this transposition nonetheless appears relatively straightforward.

Furthermore, the main advantage of the method lies, once again, in its portability : the analyses related to the bytecode and those specific to the target platform are distinct [HK07]. Application developers therefore do not need to use specific annotations to compute the WCET of an OCaLustre program. Such annotations (for example, regarding the number of iterations of a loop) can, however, be used by the platform developer to provide a table of instruction and primitive costs. Once created, this table is sufficient for analyzing the bytecode of any OCaLustre program. The portability of the method thus implies the portability of the *Bytecrawler* tool : for the same program, simply providing a cost table measured for another microcontroller makes it possible to compute the program's WCET on that platform, without even recompiling it.

The method presented in this chapter could certainly benefit from additional static analyses, for example symbolic execution [BFL18], or concolic execution [Sen07] (a hybrid approach combining, in a way quite close to our method, concrete and symbolic values), which would allow unreachable execution paths to be excluded. However, this method is above all an illustration of an analysis leveraging a common representation (bytecode), rather than a technique aiming at the most precise possible estimation of a program's maximal execution time.

Finally, it is worth noting that the method presented in this chapter could easily be extended to compute other measures beyond execution time : by simply changing the definition of the cost function (and thus the table representing it), it would be possible to evaluate other criteria, such as estimating the maximum stack size of a synchronous program, or even the maximum amount of heap-allocated values during program execution. These considerations regarding program memory footprint will be crucial in the next chapter, where we will detail different measures for evaluating the performance of the main solutions presented in this dissertation.

7 Performance of OMicroB and OCaLustre

This chapter presents and analyzes several measurements aimed at evaluating the efficiency of the software solutions described in this document. We first present a set of benchmarks carried out on the OMicroB virtual machine using programs designed to test its capabilities with respect to various aspects of the OCaml language. We then analyze the performance of OCaLustre in particular through an evaluation of the memory usage of a complete OCaLustre program, which we compare with an equivalent program written in the Lucid Synchrone language. Throughout the chapter, we discuss the results of these measurements in detail and place them in perspective relative to the performance of existing solutions.

7.1 OMicroB Performance Benchmarks

In this section, we perform several measurements to evaluate the performance of the OMicroB virtual machine. These measurements, carried out with simple test programs, concern both the speed and memory consumption of the virtual machine.

7.1.1 Methodology

OMicroB is a highly configurable virtual machine. In fact, the user can specify several concrete aspects of the machinery responsible for executing OCaml bytecode. This configurability makes it possible to adapt the virtual machine precisely to the hardware on which it runs. The adjustment of the various aspects of the virtual machine is performed using specific compilation options, provided in addition to the command used to compile an OCaml program with OMicroB :

```
omicrob <options> <file.ml>
```

In this chapter, we make use of the following compilation options :

— The option `-arch <n>` selects the word length of OCaml values in the representation used by OMicroB. This length can be 16, 32, or 64 bits. A 16-bit representation, sufficient for the vast majority of programs, reduces memory usage.

— The option `-gc <algorithm>` allows the user to choose the garbage collection algorithm. This can be the *Stop and Copy* algorithm (`SC`) described in Chapter 2, or a *Mark and Compact* (`MC`) algorithm.

— The option `-no-shortcut-initialization` disables the optimization that performs partial evaluation of the OCaml program up to its first input/output primitive.

— The option `-stack-size <n>` sets the size (in words) of the stack used by the virtual machine to execute the OCaml program bytecode. By default, this size is set to 64 words, but some programs may require a larger stack. Stack size directly affects the virtual machine's memory usage.

— The option `-heap-size <n>` sets the number (in words) of addressable values in the heap. By default, this number is 256 words. Like stack size, heap size directly affects the memory consumption of the OCaml program. The chosen garbage collection algorithm influences the actual RAM usage for the heap : indeed, because of its use of two spaces, the *Stop and Copy* algorithm requires reserving a memory section that is twice as large as that of a *Mark and Compact* garbage collector.

— Finally, the option `-no-clean-interpreter` disables the optimization that produces a virtual machine containing only the code for handling the bytecode instructions used by the program.

Because of their common use in both hobbyist and professional projects, we use throughout this section a version of OMicroB intended to run on microcontrollers of the AVR family. These microcontrollers, often embedded in *Arduino* boards, have very limited memory capacities (notably only a few kilobytes of RAM), which highlight the importance of OMicroB optimizations and demonstrate the feasibility of executing OCaml programs on hardware with (very) limited resources. Our measurements are carried out on an ATmega2560 microcontroller, featuring a computing power of 16 MIPS, 256 kilobytes of flash memory, and 8 kilobytes of RAM. Running the command `omicrob <file>.ml` generates two notable files : a first file `<file>.avr`, the executable program intended to be uploaded to the microcontroller, and a second file `<file>.elf`, the result of compilation with the native `gcc` compiler. The latter allows simulation on a PC of the execution of the produced programs.

The two main aspects measured in this chapter are :

— **Size criteria** : in particular, the size of the generated executables and their memory consumption are crucial in this application domain where hardware memory resources are highly constrained. We measure the memory usage of the generated AVR programs using the `avr-size` command, a variant of the UNIX `size` command that displays the size of the different memory sections used by the program. In particular, `avr-size` provides the additional formatting option `-C`, which groups memory sections into two categories : the amount of flash memory used for the program, and the amount of RAM used by the program.

To illustrate, Figure 7.1 shows the output of the `avr-size` command applied to a program `prog.avr`[1]. It indicates that this program uses 6968 bytes of flash memory (2.7% of the flash available on an ATmega2560) and 7104 bytes of RAM (86.7% of the RAM available on this microcontroller).

```
$ avr-size -C prog.avr --mcu=atmega2560
AVR Memory Usage
----------------
Device: atmega2560

Program:    6968 bytes (2.7% Full)
(.text + .data + .bootloader)

Data:       7104 bytes (86.7% Full)
(.data + .bss + .noinit)
```

FIGURE 7.1 – Example output of the `avr-size` command

— **Speed criteria** : these illustrate the temporal performance of the virtual machine. Speed measurements are carried out both on *.elf* programs corresponding to the PC simulation of OMicroB (to

1. The option `--mcu` specifies the microcontroller model.

evaluate OMicroB's performance relative to the standard virtual machine implementation), and via physical execution time measurements of programs running on a microcontroller. Simulated program executions are measured using the UNIX `time` command, which reports the execution duration of a program. From these measurements, and using the information provided by the simulation regarding the total number of executed instructions, it is possible to compute an average speed of OMicroB in terms of instructions per second. Note, however, that this speed may vary depending on the complexity of the executed bytecode instructions : processing a rich instruction (such as the `CLOSURE` instruction for closure creation) may take significantly longer than processing a basic instruction (such as `PUSH`, which adds an element to the stack). In this sense, the measured speed can only serve as an approximate indication of OMicroB's capabilities, which may vary considerably from one program to another.

These two evaluation criteria strongly depend on the compilation options described earlier. For example, the RAM usage of programs varies with the chosen stack and heap sizes, while the execution speed of a program can depend on the chosen architecture : for instance, a microcontroller may take longer to load and process data represented on 32 bits than on 16 bits. Moreover, the more frequent activation of garbage collection when using a small heap increases the execution time of OCaml programs, since part of their execution is then dedicated to cleaning unused memory in the heap.

7.1.2 Test Programs

The various measurements presented in this section are performed on simple OCaml programs, each designed to test fundamental features of the language and its virtual machine. The source code of these programs is available online [⚓1], and we briefly describe the operation of each of them below :

1. The program `apply.ml` performs one thousand successive applications of the function $(\lambda f\, x.f\ (f\ x))$, which represents the Church encoding of integers. This program tests the mechanism of higher-order function application.
2. The program `fibo.ml` computes the first ten terms of the Fibonacci sequence one hundred thousand times. This program tests recursive function application as well as the use of basic arithmetic operations.
3. The program `takc.ml` computes the result of the Takeuchi function one thousand times, with parameters 18, 12, and 6. This program manipulates triples, testing both the garbage collector and recursive function application.
4. The program `oddeven.ml` computes the parity of integers between 0 and 100 ten thousand times. This program tests the application of mutually recursive functions.
5. The program `floats.ml` applies trigonometric functions (`sin` and `cos`) to floating-point numbers ten million times. It tests floating-point representation and the performance of operations on floats.
6. The program `integr.ml` computes ten thousand times the integral $\int_0^{10}(x^2 + 2x + 10)\, dx$ with a step of 0.01. This program tests floating-point representation as well as the dynamic generation of functional values.
7. The program `eval.ml` is a Boolean formula evaluator that computes ten thousand times the value of ten Boolean formulas. It tests the definition of algebraic data types, pattern matching, and the use of the `Stack` module, which represents a stack data structure.

8. The program `sieve.ml` computes the prime numbers less than 50 ten thousand times using the Sieve of Eratosthenes algorithm. This program induces list creation and thus tests the garbage collection capabilities.
9. The program `objet.ml` tests OCaml's object representation : it defines a `point` class, generates one hundred thousand objects of this class, and for each computes ten times its symmetric with respect to the origin by calling the `sym` method.
10. The program `functor.ml` tests the mechanism of parameterized modules (or *functors*) in OCaml. It generates a module representing a set of integers using the `Set.Make` functor from the standard library, then adds to this set, ten thousand times, the values contained in a list of 50 consecutive integers.
11. The program `bubble.ml` implements the bubble sort algorithm using arrays. It generates an array of 60 elements populated with decreasing integers and applies bubble sort to this array ten thousand times. This program tests the implementation of arrays, access to their values, and their updates.
12. The program `jdlv.ml` implements Conway's Game of Life on a 10×10 matrix. It computes the first ten thousand configurations of the Game of Life starting from a state representing an oscillator. This program, which uses mutable record fields and matrices, tests their implementation.
13. The program `share.ml` defines a list filtering function that uses OCaml's exception mechanism to avoid unnecessary copying of the original list elements into the resulting list. It filters the list of natural integers from 0 to 50 one hundred thousand times using the function ($\lambda x.x > 25$). This program pushes many exception handlers and thus tests the exception-handling mechanism in OCaml.
14. The program `abrsort.ml` performs binary search tree sorting of a set of natural integers between 0 and 100 ten thousand times. It tests OMicroB's performance on a recursive data structure.
15. Finally, the program `queens.ml` solves the n-queens problem ten thousand times, with n queens ranging from 1 to 6. This program tests the computing power of the virtual machine on a non-trivial algorithmic example.

The measurements on these programs were carried out in two phases : the programs were first tested on a PC by compiling OMicroB with the *gcc* compiler. In a second phase, the same tests were executed on an AVR ATmega2560 microcontroller.

7.1.3 Execution on PC : Results and Interpretation

The programs discussed in this section were compiled with the OMicroB option `-no-shortcut-initialization`, in order to avoid triggering OMicroB's partial evaluation mechanism, ensuring that all computations are performed at runtime, since these tests make no use of input/output primitives.

All measurements were performed using a 16-bit representation of OCaml values, the *Stop and Copy* garbage collection algorithm, a heap size of 2500 words, and a stack size of 500 words. The computer used was equipped with an Intel Core i5-8259U quad-core processor, with a 64-bit architecture and a clock speed of 2.3 GHz.

Execution speed : The tests compare the execution speed of the OMicroB virtual machine (in number of bytecode instructions per second) with that of the standard ZAM bytecode interpreter, named *ocamlrun*. The latter was used in its standard configuration : 64-bit encoded values, a stack of 256,000 words, and a minor heap of 2 megabytes. The results of the measurements carried out with the configuration described above are presented in Table 7.1. In particular, the column labeled Ratio represents the ratio between the execution time of a program with OMicroB and its execution time with ocamlrun : a smaller value is therefore preferable. Additional results obtained with the *Mark and Compact* GC, as well as with 32-bit and 64-bit data representations, are available in Appendices C.1, C.2, and C.3.

Name	**Execution time with ocamlrun** (seconds)	**Execution time with OMicroB** (seconds)	**Ratio**	**Execution speed with OMicroB** (million instr. bytecode/sec)	**Number of OMicroB GC triggers**
`apply`	1.20	2.14	1.78	306.33	58
`fibo`	0.55	1.14	2.07	428.82	0
`takc`	1.05	3.07	2.92	383.31	226000
`oddeven`	0.28	0.60	2.14	614.68	0
`floats`	0.56	1.05	1.87	219.46	0
`integr`	0.04	0.12	3.00	222.16	42
`eval`	0.04	0.07	1.75	431.42	2352
`sieve`	0.04	0.07	1.75	486.00	3333
`objet`	0.10	0.17	1.70	389.77	11765
`functor`	0.24	0.60	2.50	392.77	30003
`bubble`	0.93	1.55	1.66	461.49	35
`jdlv`	0.36	0.75	2.08	457.64	6666
`share`	0.11	0.25	2.27	480.24	699
`abrsort`	0.47	1.22	2.59	321.27	119924
`queens`	1.35	3.35	2.48	413.54	149999

Table 7.1 – Performance measurements of OMicroB programs (on PC)
OMicroB options : -arch 16 -gc SC -stack-size 500 -heap-size 2500

From the measurements carried out on PC, it appears that OMicroB's execution speed can in some cases be about 2 to 3 times slower than ocamlrun, the standard bytecode interpreter. These results remain quite reasonable, as OMicroB's performance is still within an expected order of magnitude, and our optimizations primarily target memory consumption rather than execution speed. This performance difference can be explained by several factors. Notably, the representation of the program as a byte-by-byte array of C values induces an obvious overhead, compared to ocamlrun loading the program into RAM and directly reading values whose size corresponds to the processor's architecture (32 or 64 bits). Moreover, memory constraints differ between OMicroB (here : 2500 words for the heaphalf of which is actually available because of the *Stop and Copy* garbage collector) and ocamlrun (whose minor heap is by default 2 megabytes on a 64-bit platform). This leads to more frequent triggering of OMicroB's garbage collector, slowing down program execution.

For illustration, the graphs in Figure 7.2 show the execution times of the `takc.ml` and `abrsort.ml` programs as a function of different heap sizes and GC algorithms (with a 16-bit representation of values) :

the general trend is a decrease in execution time as the available heap size increases, stabilizing once the program consumes less memory than is available.

Finally, the `floats` test highlights the advantage of our immediate representation of floating-point values : in this test, OMicroB is almost as fast in 64-bit mode, and even slightly faster in 32-bit mode, than ocamlrun (see Tables C.2 and C.3 in the appendices). The absence of heap allocation for floats both reduces GC triggers and limits the number of indirections induced by the standard representation of floats, which are typically encapsulated and allocated on the heap.

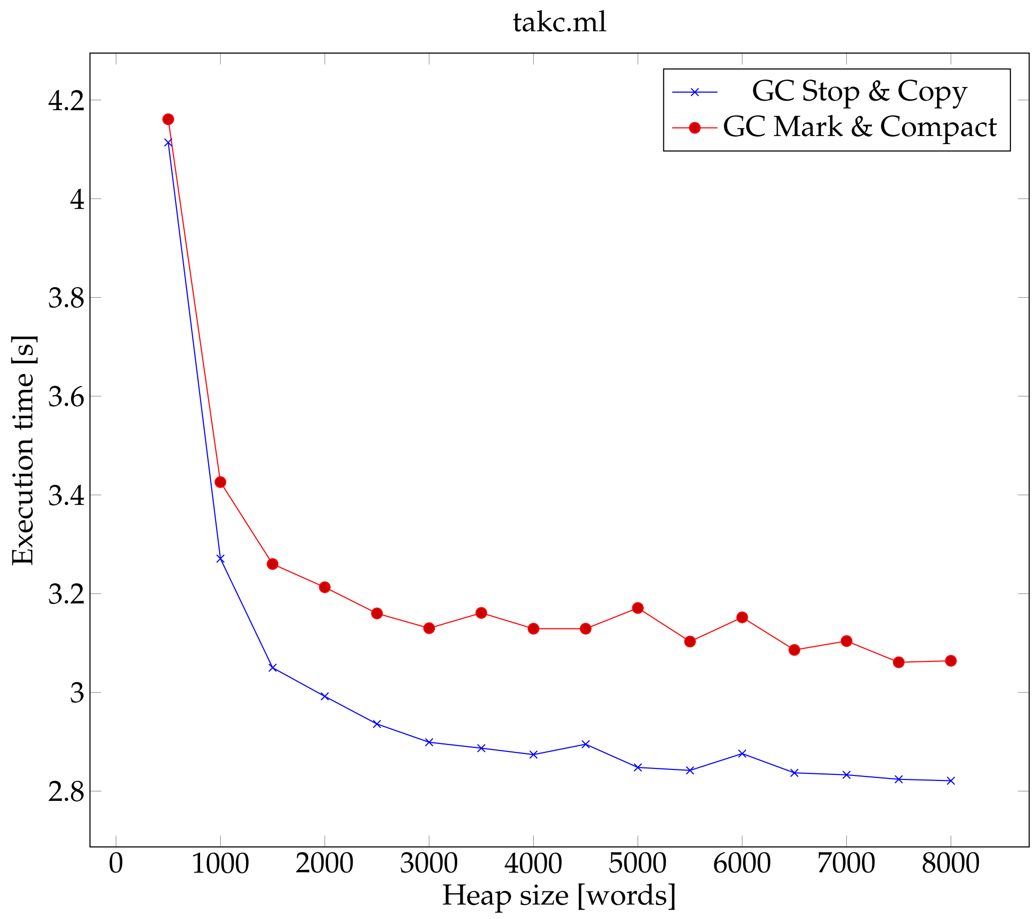
7.1.4 Execution on Microcontroller : Results and Interpretation

In the following, we present and interpret the performance measurements obtained when running the test programs on a microcontroller. The target is an AVR ATmega2560 microcontroller, with 256 KB of flash memory, 8 KB of RAM, and a clock rate of 16 MIPS. The programs are compiled with the `-no-shortcut-initialization` option, using a 16-bit representation of OCaml values, a maximum stack size of 500 words, a heap size of 2500 words, and the *Stop and Copy* garbage collector. Note that, due to the relative slowness of a microcontroller compared to a modern PC, the number of computations performed in these tests was reduced by a factor of one thousand compared to the PC tests, in order to keep measurement durations reasonable. For example, the `queens.ml` program now solves the n-queens problem only ten times (with n from 1 to 6).

Name	**Execution time** (seconds)	**Speed** (thousands of instr./second)	**Number of GC triggers**
`apply`	3.579	183.191	0
`fibo`	1.984	246.473	0
`takc`	5.475	214.951	228
`oddeven`	1.334	276.536	0
`floats`	7.482	30.812	0
`integr`	0.745	35.910	0
`eval`	0.143	212.048	2
`sieve`	0.195	174.953	3
`objet`	0.420	162.138	12
`functor`	1.351	192.904	33
`bubble`	3.762	190.169	0
`jdlv`	2.006	171.273	6
`share`	0.572	211.891	0
`abrsort`	2.677	147.965	124
`queens`	6.598	209.987	154

TABLE 7.2 – Execution speed measurements of OMicroB programs (on ATmega2560)
OMicroB options : -arch 16 -gc SC -stack-size 500 -heap-size 2500

Execution speed : Table 7.2 shows the execution time results for the programs on the 16-bit version of the virtual machine. Timing measurements were performed using the OCaml function `Avr.millis : unit -> int`, which returns the number of milliseconds elapsed since the program was launched on the



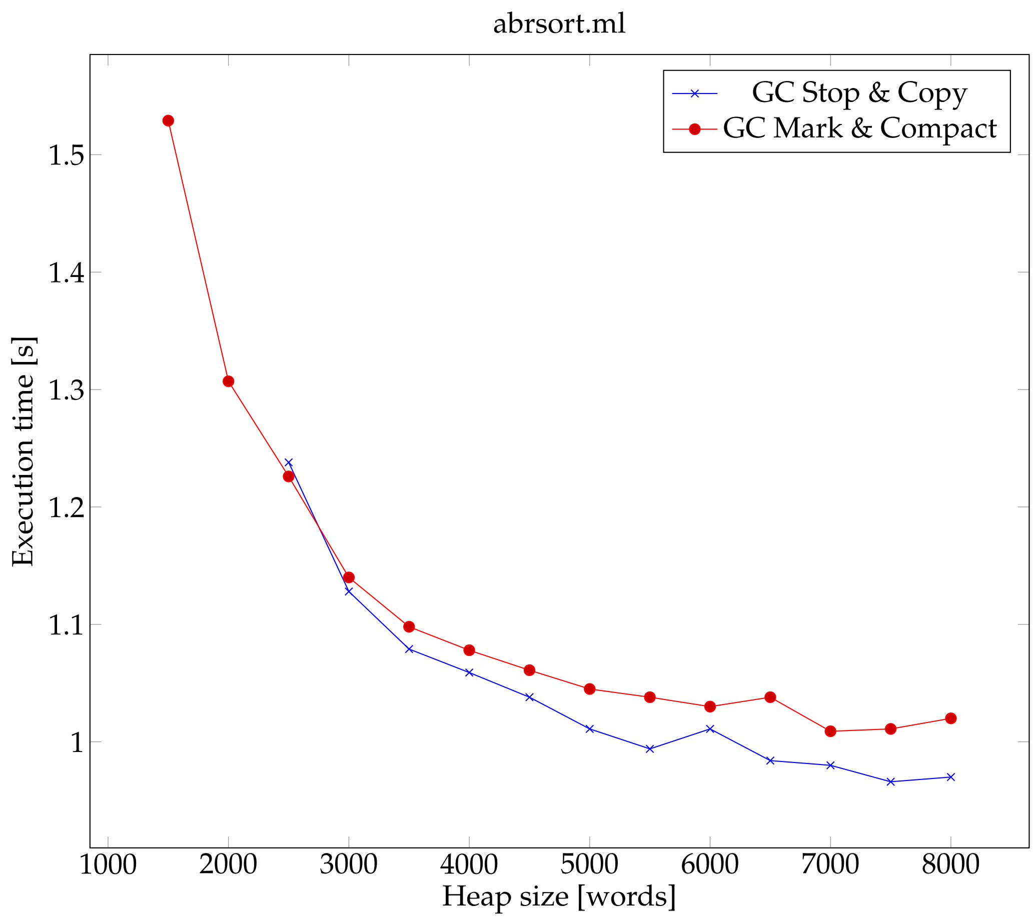


Figure 7.2 – Influence of heap size on execution time (on PC)

microcontroller. The difference between the value returned by this function at the end of the program and the value returned at the start provides an estimate of its execution time. This value is then transmitted, via a serial communication protocol, to the personal computer connected to the microcontroller.

The measurements allow us to evaluate the execution speed of OMicroB on AVR at approximately 200,000 bytecode instructions per second. These results depend on many factors, such as the capabilities and level of optimization of the compiler used (here, the option applied to `avr-gcc` is `-O2`), the nature of the executed instructions (whose execution times vary according to their level of complexity), and hardware-specific characteristics. In particular, the `floats.ml` and `integr.ml` tests, which manipulate floating-point numbers, exhibit significantly lower execution speeds. This slowdown is explained by the specifics of the hardware used : on an AVR microcontroller, floating-point operations are performed in software, and are thus much slower than on a PC, where such operations are handled directly by the processor's arithmetic unit. Furthermore, the non-standard representation of floats in the 16-bit version of the virtual machine imposes additional conversions during program execution.

Table C.4, in the appendix, presents the measurements for the same programs using a 32-bit version of the virtual machine. As expected, the measured speed is slower by a factor of roughly 2.

Comparison with OCaPIC : We also compare these results with executions of a subset of the test programs on OCaPIC, the OCaml virtual machine designed for PIC 18 microcontrollers. In particular, we ran these programs on a PIC 18f4620 microcontroller, equipped with 64 KB of flash memory, 4 KB of RAM, and a clock rate of 10 MIPS. Table 7.3 presents the execution time results for these programs[2]. Since OMicroB has not yet been fully ported to PIC 18, these times are compared with those obtained on the ATmega2560, by extrapolating its computing speed to match the 10 MIPS of the PIC 18f4620. In this sense, the performance comparison between OCaPIC and OMicroB is necessarily approximate, intended only to provide a rough idea of the relative performance order of magnitude of the two virtual machines. From these measurements, it follows that OMicroB is about 3.6 times slower (at equal clock frequencies) than OCaPIC. It is important to note that the latter benefits from a bytecode interpreter and garbage collector written in assembly, leveraging PIC-specific optimizations, which may account for its performance. Moreover, hardware-specific factors also affect program execution speed : for example, a PIC requires only 2 clock cycles to read a byte from flash memory, compared to 12 cycles on an AVR microcontroller.

In addition, preliminary experiments to port OMicroB to PIC 18 highlighted the influence of compiler quality on the execution speed of OCaml programs : using the `xc8` compiler provided by the manufacturer results in execution speeds about 7 times slower on OMicroB than on OCaPIC, for the same program and on the same microcontroller. This relative slowness is partly explained by the poor optimization of the generated machine code. For instance, the *switch* statement in the bytecode interpreter, whose role is to distinguish instructions based on their opcode in order to execute them, was transformed by `xc8` into a cascade of nested conditional branch instructionsobviously impacting execution speed.

Nevertheless, the temporal performance of OMicroB remains overall quite acceptable, and the fact that OMicroB is portable is an advantage that must be taken into account. Thus, given the portability of the solution, OMicroB's performance demonstrates the viability of using a generic virtual machine. In Chapter 8, we will present a set of applications that demonstrate the responsiveness of the virtual machine, as well as its suitability for executing embedded programs.

2. The other programs could not be used on this hardware due to excessive memory requirements.

Name	**Execution time with OMicroB on ATmega2560 at 10 MIPS** (seconds)	**Execution time with OCaPIC on PIC 18F4620 at 10 MIPS** (seconds)	**Ratio OMicroB/OCaPIC**
`apply`	5.726	1.632	3.51
`fibo`	3.174	0.860	3.69
`takc`	8.76	2.535	3.46
`oddeven`	2.134	0.657	3.25
`integr`	1.192	0.252	4.73
`eval`	0.229	0.064	3.58
`sieve`	0.312	0.084	3.71
`bubble`	6.030	1.793	3.36
`share`	0.915	0.251	3.65

Table 7.3 – Comparison of execution times with OCaPIC
OMicroB options : -arch 16 -gc SC -stack-size 500 -heap-size 2500

Comparison with MicroPython : To give an indication of OMicroB's performance relative to existing and frequently used solutions, we carried out a few comparative measurements between some of our OCaml example programs and their equivalents written in Python. The latter were executed by the MicroPython implementation on a *pyboard* (version 1.1) equipped with an STM32F405RG microcontroller, featuring 1024 KB of flash memory, 192 KB of RAM, and a processor with a clock speed of 168 MHz. As in the comparison with OCaPIC, we extrapolate the results of the measurements obtained on the pyboard in order to compare them with OMicroB's measurements on an ATmega2560 at 16 MHz.

The comparison between MicroPython and OMicroB can only provide a broad indication of their relative performance, given the significant differences between programs written in OCaml and those written in Python. Indeed, programming in a camel-like functional style makes heavy use of recursive functions and data structures, whereas recursion in Python is rather limited : for example, Python lacks tail-call optimization, which can transform certain recursive functions into iterative ones. As a result, many of the test programs presented in this section are incompatible with execution under MicroPython, as they quickly exhaust the call stack depth. On a pyboard, this stack is furthermore quite shallow, with a depth of only 63 frames.

We therefore present, in Table 7.4, the measurement results obtained on a subset of our test programs corresponding to those that can run under MicroPython without requiring major structural modifications. It should be noted that the `bubble` test, which implements bubble sort, uses Python *lists* in our Python version[3], since arrays from the Numpy library are not available in MicroPython. The `jdlv` test, on the other hand, was implemented using objects.

The results of these measurements are very promising : OMicroB is consistently faster, at equal clock frequencies. Notably, the `fibo` test (likely due to the large number of function calls) shows a clear speed advantage, and OMicroB performs significantly better on the `takc` test, which dynamically allocates many tuples during execution (and triggers numerous garbage collections). However, OMicroB also demonstrates good performance on imperative and object-oriented tests. It is worth noting that the

3. Accessing and updating an element in such lists still has $\mathcal{O}(1)$ complexity, just like OCaml arrays.

Name	**Execution time with OMicroB on ATmega2560 at 16 MHz** (seconds)	**Execution time with MicroPython on pyboard (scaled to 16 MHz)** (seconds)	**Ratio MicroPython/OMicroB**
`fibo`	1.984	5.481	2.76
`takc`	5.475	155.620	28.42
`bubble`	3.762	4.420	1.17
`jdlv`	2.006	3.024	1.5
`objet`	0.420	0.630	1.5

Table 7.4 – Comparison of execution times with MicroPython
OMicroB options : -arch 16 -gc SC -stack-size 500 -heap-size 2500

superior results for the `fibo` and `takc` tests do not stem from OCaml's tail-call optimization, since it is not applied in `fibo`, and in `takc` only the outer recursive call of the Takeuchi function is optimized.

Program and Interpreter Size : Table 7.5 shows the memory footprint of each program on the flash and RAM of the microcontroller. The almost constant RAM footprint across programs is due to the fact that the stack and heap sizes are fixed at compile time : static arrays are generated to represent these memory sections, and their size cannot change during execution. It is worth noting that the flash memory usage of the `objet` and `functor` programs is relatively large. In the first case, this is because the entire mechanism required to create and manipulate objects is imported into the program's bytecode. While this incurs a significant flash memory cost, it remains stable : a program that manipulates ten times more objects and classes does not have ten times the memory footprint. In the case of the functor program, the extra flash memory consumption is related to the fact that *ocamlclean* is unable to statically detect dead code inside a module produced by a functor application. As a result, in our test, the generated bytecode contains the full execution code of the module produced by applying the `Set.Make` functor, even though only a small subset of its functions are actually used.

The interpreter of the virtual machine, along with its runtime library, also has a significant weight in these measurements. The size of the interpreter and garbage collector can be estimated by measuring the flash memory footprint of a program that only computes the *unit* value. The bytecode of such a program reduces to the single `STOP` instruction. Since the optimization that removes unused bytecode handlers from the interpreter is enabled by default, the resulting executable has a very small flash footprint : in a 16-bit version of OMicroB, the `avr-size` command reports 1240 bytes in flash. This corresponds to the size of the smallest possible OCaml program compiled with OMicroB.

In contrast, when the same program is compiled with OMicroB's `-no-clean-interpreter` option, the executable's flash size is 21340 bytes. Consequently, the interpreter plus the garbage collector (which is not included when all allocation-related bytecode handlers are removed) occupy about 20 KB. A program reduced to the single `CLOSURE` instruction, which allocates a closure (and thus requires the garbage collector), has a size of 3110 bytes with the interpreter cleaning enabled. This indicates that the footprint of the *Stop and Copy* garbage collector is around 2 KB. Equivalent measurements with the *Mark and Compact* collector show a footprint of about 3 KB.

Name	Flash footprint (bytes)	RAM footprint (bytes)
`apply`	7626	6116
`fibo`	8614	6116
`takc`	8568	6114
`oddeven`	8230	6112
`floats`	10536	6128
`integr`	10870	6122
`eval`	13420	6239
`sieve`	9550	6119
`objet`	24120	6327
`functor`	20162	6241
`bubble`	11926	6217
`jdlv`	11772	6149
`share`	11122	6135
`abrsort`	13600	6219
`queens`	10704	6139

Table 7.5 – Program size measurements for OMicroB (on ATmega2560)
OMicroB options : -arch 16 -gc SC -stack-size 500 -heap-size 2500

In a 32-bit configuration of the virtual machine, the interpreter and garbage collector occupy about 28 KB of flash, while in 64-bit they occupy about 40 KB. These measurements show that without the optimization brought by compiling a *customized interpreter*, programs using this virtual machine would quickly reach the flash memory limits of the targeted microcontrollers (for example, an ATmega32u4 has only 32 KB of flash memory). Thanks to this optimization, the test programs consume on average only about ten kilobytes of flash memory, which corresponds to the size of the interpreter specialized for each program, its runtime library, and the program bytecode itself, which also resides in flash. This optimization therefore provides significant benefits and enables OMicroB to run on hardware where the full interpreter could not otherwise fit.

It should be noted that a program using all bytecode instructions of the virtual machine would, of course, also import the entire bytecode interpreter. However, such cases are extremely rare, and the programs presented in this thesis illustrate that most OCaml programs use only a relatively small (though variable) subset of the available bytecode instructions.

7.2 OCaLustre Performance Measurements

In this section, we evaluate the performance of the OCaml code produced by OCaLustre and its suitability with respect to the hardware constraints of the microcontrollers targeted in this thesis. To do so, we design an example OCaLustre program that performs several computations, and we measure both the execution speed and the memory footprint of the program when executed within OMicroB.

This example is a *parallel ripple-carry adder*, which computes the binary sum of 8-bit words. The adder, illustrated in Figure 7.3, consists of eight independent elements, each responsible for adding two single-bit values. These 1-bit adders compute the sum of two bits as well as any carry, and are called *full* *adders* because they also take as input the potential carry generated by the adder responsible for the less significant bit. The sum of two bytes ($a_7a_6a_5a_4a_3a_2a_1a_0$ and $b_7b_6b_5b_4b_3b_2b_1b_0$) is thus obtained by adding each pair of bits (a_i, b_i) while propagating the carry generated by each 1-bit adder to the next one.

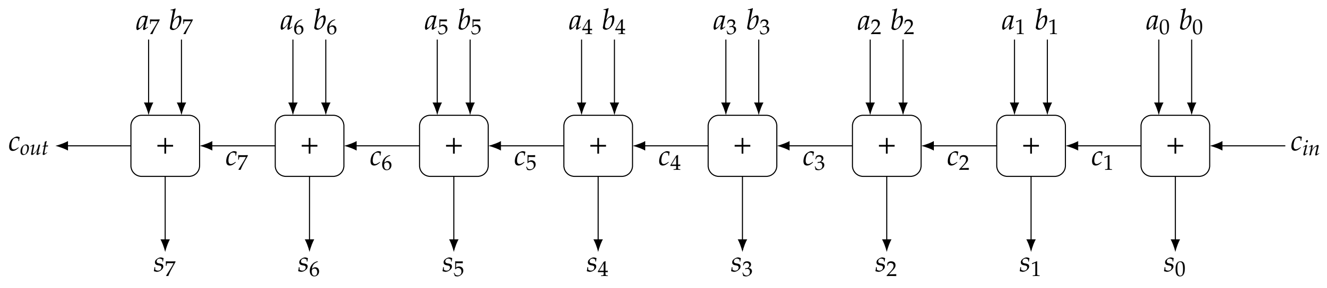


FIGURE 7.3 – 8-bit ripple-carry adder

7.2.1 A Binary Adder in OCaLustre

A 1-bit adder is implemented in OCaLustre as a node named `fulladder`, whose definition closely follows the standard representation of an adder in the form of a logic-gate diagram, as illustrated in Figure 7.4.

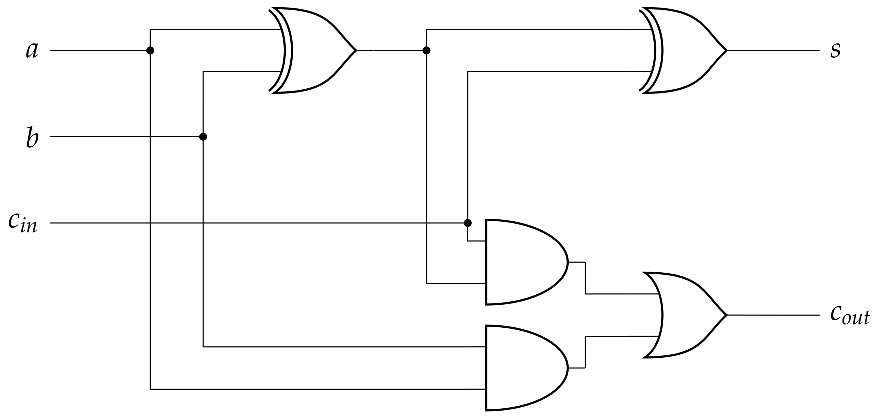


FIGURE 7.4 – Circuit of a full adder

The code of the `fulladder` node together with the OCaml function `xor` used to compute the exclusive-or between two booleans is as follows :

```
let xor a b = if a then not b else b

let%node fulladder (a,b,cin) ~return:(s,cout) =
  x = call xor a b;
  s = call xor x cin;
  and1 = (x && cin);
  and2 = (a && b);
  cout = (and1 || and2)
```

This node computes the sum `s` of two bits `a` and `b` together with an input carry `cin`, as well as the potential output carry `cout`. The bit values are represented here as booleans.

From a single fulladder node, it is then straightforward to define an adder capable of computing the sum of two 2-bit values in the form of the node twobits_adder, whose role is simply to chain the computations of two adders :

```
let%node twobits_adder (c0,a0,a1,b0,b1) ~return:(s0,s1,c2) =
  (s0,c1) = fulladder (a0,b0,c0);
  (s1,c2) = fulladder (a1,b1,c1)
```

This node therefore computes three booleans corresponding to the sum s_0s_1 of the two values a_0a_1 and b_0b_1, as well as the possible carry c_2. Following the same principle, we can then define a 4-bit adder, and finally an 8-bit adder :

```
let%node fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3) ~return:(s0,s1,s2,s3,c4) =
  (s0,s1,c2) = twobits_adder (c0,a0,a1,b0,b1);
  (s2,s3,c4) = twobits_adder (c2,a2,a3,b2,b3)

let%node eightbits_adder (c0,a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7)
                       ~return:(s0,s1,s2,s3,s4,s5,s6,s7,c8) =
  (s0,s1,s2,s3,c4) = fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3);
  (s4,s5,s6,s7,c8) = fourbits_adder (c4,a4,a5,a6,a7,b4,b5,b6,b7)
```

This example perfectly illustrates the compositionality inherent in the synchronous approach of OCaLustre : the definition of a node can be reused and combined by another node, which in turn can be composed with yet another one, making it straightforward to build increasingly complex applications without the developer having to worry about causal relationships between the different components of the program.

7.2.2 Results

From the eightbits_adder node defined above, we construct an OCaLustre program that computes one hundred thousand times the sum $0b11111111 + 0b11111111$. This example is then executed, in the same way as the previous tests, using the OMicroB virtual machine on a PC.

Table 7.6 shows the performance measurements obtained for this program. In particular, the memory footprint of OCaLustre is very small : a stack of 80 words and a heap of 400 OCaml values are sufficient to execute the program, leading to a total RAM consumption of only 1069 bytes and a flash memory consumption of 7668 bytes.

	Execution time	**Speed**	**Number of GC**
Nom	**(seconds)**	**(millions of instr./sec)**	**triggers**
adders.ml	0.54	138.01	550054

TABLE 7.6 – Measurements of the execution speed of the adder with OMicroB (on PC)
OMicroB options : -arch 16 -gc SC -stack-size 80 -heap-size 400

In terms of execution speed, the performance of OCaLustre programs in OMicroB is good, but nevertheless slowed down by the large number of garbage-collector invocations. As illustrated in Figure

7.5, increasing the heap size drastically reduces the number of GC triggers, thereby accelerating program execution.

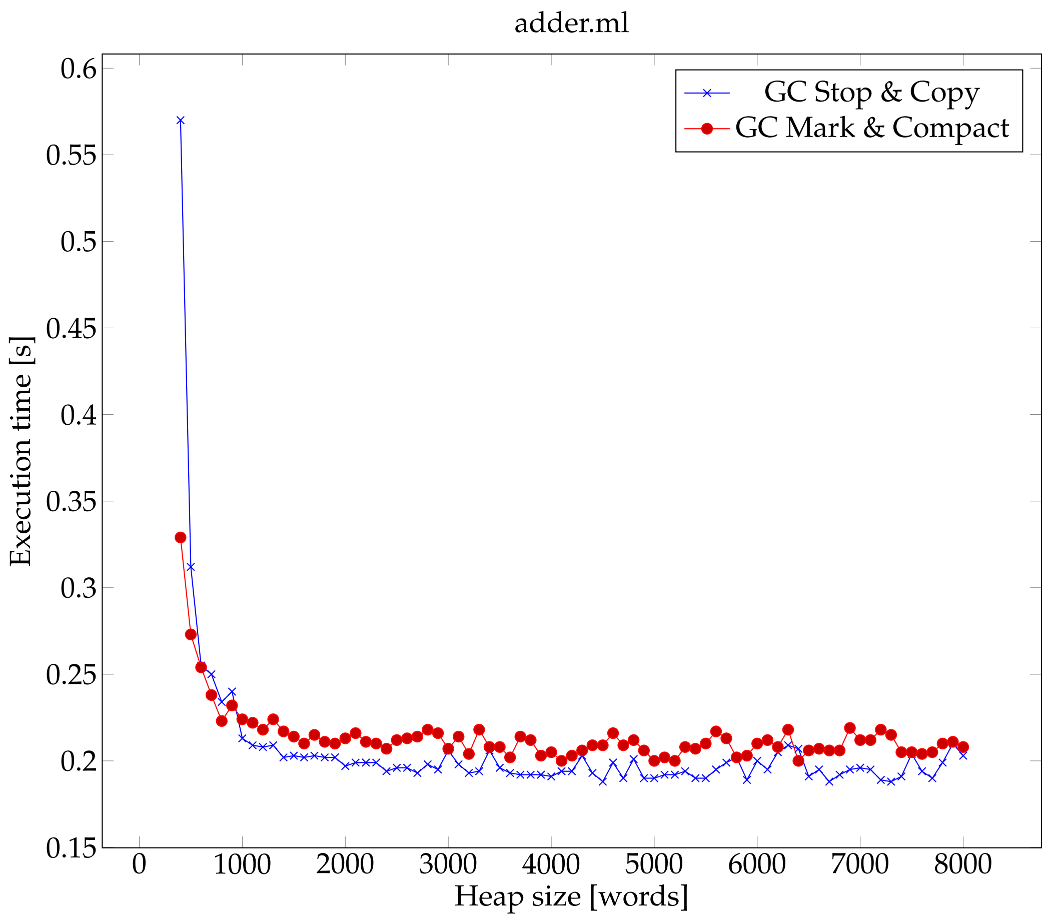


Figure 7.5 – Execution time of the adder as a function of heap size (on PC)

The -na compilation option of OCaLustre generates an imperative style of code optimized for critical embedded contexts, in the sense that it does not allocate new values during a synchronous instant. As a result, it avoids triggering the garbage collector during the execution of the synchronous program. Consequently, the execution speed of the adder is greatly improved when this option is enabled (Table 7.7).

Name	**Execution time (seconds)**	**Speed (millions of instr./sec)**	**Number of GC triggers**
`adder-noalloc.ml`	0.21	383.14	0

Table 7.7 – Execution speed of the adder without heap allocations during a synchronous instant
OMicroB options : -arch 16 -gc SC -stack-size 80 -heap-size 400

Comparison with Lucid Synchrone : We now propose to compare the performance of this program with that of an equivalent program written in the Lucid Synchrone language. Although the development of Lucid Synchrone has now been discontinued, its compiler (version 3, dating from April 2006) remains available on the project's website [⚓19]. This compiler produces OCaml code, making it possible to compare it directly with OCaLustre.

The goal of this comparison is mainly to validate the relevance of using OCaLustre for programming resource-constrained hardware, by verifying that synchronous programs written in OCaLustre could not simply have been written in Lucid Synchrone while maintaining equivalent performance. Of course, Lucid Synchrone is a rich extension of Lustre, with greater expressiveness than OCaLustre, and this expressiveness is also reflected in the complexity of the generated code. The comparison presented here should therefore not be regarded as a benchmark of the raw performance of the two languages, but rather as an illustration of the space savings provided by our syntactic extension for the kind of programs we target, due to its relative simplicity.

Figure 7.8 reports the measurements obtained on the Lucid Synchrone program `adder.ls` (whose code very close to the syntax of OCaLustre is given in Appendix B), as well as on the OCaLustre program in its standard version (`adder.ml`) and in its version optimized for critical embedded systems (`adder-na.ml`). Since the memory requirements to run the Lucid Synchrone program are higher, the measurements were carried out with a stack of 150 words and a heap of 600 words.

Program	**Execution time** (seconds)	**Speed** (millions instr./second)	**Number of GC triggers**
`adder.ls`	0.79	222.87	440042
`adder.ml`	0.25	298.11	110010
`adder-na.ml`	0.19	423.47	0

Table 7.8 – Performance measurements of the adder program with OCaLustre and Lucid Synchrone (on PC)
OMicroB options : -arch 16 -gc SC -stack-size 150 -heap-size 600

It follows from these measurements that, on this small program and under equal virtual machine configurations, the performance of Lucid Synchrone is three to four times lower than that of OCaLustre on a PC. On an ATmega2560, the execution speed of Lucid Synchrone is between four and six times lower than that of OCaLustre programs[4] (Table 7.9). Due to the expressiveness of the language, the OCaml code generated by Lucid Synchrone is considerably larger and more resource-demanding than that generated by OCaLustre. Figure 7.6 shows, for the `twobits_adder` node, the OCaml code produced by Lucid Synchrone compared with that produced by OCaLustre compilation. In particular, the Lucid Synchrone version relies on polymorphic variants, which are relatively costly to represent certain values. It also generates, at each instant, several references to tuples, with the effect of filling the virtual machine's heap fairly quickly. As a result, the need for the Lucid Synchrone program to increase the heap size and to double the stack size is quite penalizing when targeting microcontrollers whose memory resources are limited to only a few kilobytes.

Measurements on the manuscript examples : Table 7.10 shows the execution of OCaLustre programs taken from most of the examples discussed in this manuscript. The name of each program corresponds to the name of its main node, which is executed one million times. The measurements confirm the performance results observed with the adder, as well as the low memory consumption of OCaLustre. The hardware resource usage of the OCaLustre extension is small : its flash memory footprint is comparable

4. As before, the number of instants computed was divided by one thousand for our measurements on the microcontroller.

Name	Execution time (seconds)	Speed (thousands of instr./second)	Number of GC	Flash size (bytes)	RAM size (bytes)
`adder.ls`	2.181	88.827	482	9250	1627
`adder.ml`	0.554	148.185	120	7552	1609
`adder-na.ml`	0.366	221.453	0	8472	1613

TABLE 7.9 – Execution speeds and sizes of Lucid Synchrone and OCaLustre programs (ATmega2560)
OMicroB options : -arch 16 -gc SC -stack-size 150 -heap-size 600

```
let twobits_adder _cl147 (_148__c0, _149__a0, _150__a1, _151__b0, _152__b1) _325
 _self_364 =
  let _self_364 = match !_self_364 with
                | ‘St_369(_self_364) -> _self_364
                | _ -> (let _368 = {_366 = ref ‘Snil_;
                                    _365 = ref ‘Snil_;
                                    _init329 = true} in
                       _self_364 := ‘St_369(_368);
                       _368) in
  let _cl339__ = ref false in
  let _338 = ref (false, false) in
  let _cl337__ = ref false in
  let _328 = ref false in
  let _340 = ref (false, false) in
   (if _cl147 then
     (_328 := (or) _325 _self_364._init329;
      _cl337__ := true;
      _338 := fulladder !_cl337__ (_149__a0, _151__b0, _148__c0) !_328 _self_364._365));
   (let (_153__s0, _154__c1) = !_338 in
    (if _cl147 then
      (_cl339__ := true;
       _340 := fulladder !_cl339__ (_150__a1, _152__b1, _154__c1) !_328 _self_364._366));
    (let (_155__s1, _156__c2) = !_340 in
     _self_364._init329 ← (&) !_328 (not _cl147);
     (_153__s0, _155__s1, _156__c2)))
```

```
let twobits_adder () =
  let fulladder1_app = fulladder () in
  let fulladder2_app = fulladder () in
  fun (c0, a0, a1, b0, b1) ->
    let (s0, c1) = fulladder1_app (a0, b0, c0) in
    let (s1, c2) = fulladder2_app (a1, b1, c1) in
    (s0, s1, c2)
```

FIGURE 7.6 – OCaml code of the `twobits_adder` node generated by Lucid Synchrone (top) and by OCaLustre (bottom)

to that of other OCaml programs, and its RAM footprint is also low. This frugal use of resources makes OCaLustre compatible with many microcontroller models, some with memory capacities of around 2 kilobytes. OCaLustre thus provides a high-level programming model that enables the development of synchronous programs with strong guarantees (regarding typingof clocks as well as of manipulated dataor the detection of causal inconsistencies within programs). These advantages are all the more important since they do not entail increased resource consumption, thereby enabling the development of embedded systems running on hardware with limited capacity.

Program name	**Execution time with ocamlrun** (seconds)	**Execution time with OMicroB** (seconds)	**Ratio**	**Execution speed with OMicroB** (millions of instr. bytecode/second)	**Number of GC runs with OMicroB**
`arith`	0.85	1.54	1.81	337.76	357142
`blinker`	0.02	0.04	2.00	521.67	0
`call_cpt`	0.06	0.11	1.83	488.14	19058
`cpt`	0.01	0.03	3.00	645.06	0
`ex_const`	0.04	0.09	2.25	518.05	17721
`ex_norm`	0.03	0.06	2.00	575.05	0
`ex_tuples`	0.06	0.11	1.83	396.31	37411
`fibonacci`	0.02	0.04	2.00	635.31	0
`fibonacci2`	0.02	0.04	2.00	660.56	0
`merge`	0.03	0.06	2.00	558.22	0
`ordo`	0.06	0.13	2.16	397.50	37411
`two_cpt`	0.03	0.08	2.66	481.79	18705
`watch`	0.06	0.13	2.16	428.58	40404
`when`	0.06	0.10	1.66	395.53	18365
`whennot`	0.03	0.05	1.66	548.65	0

Table 7.10 – Performance measurements of OCaLustre programs (on PC)
OMicroB options : -arch 16 -gc SC -stack-size 50 -heap-size 500

Chapter Conclusion

The various measurements presented in this chapter confirm that our approach, based on the use of a virtual machine, is both viable and well-suited for programming microcontrollers in a high-level language. The mere fact that OCaml programs can be executed, using a generic interpreter, on hardware with such limited resources is already a success. Moreover, the performance of the OMicroB virtual machine and of the synchronous OCaLustre extension is entirely satisfactory in terms of both execution speed and memory consumption of the generated programs : non-trivial applications can be envisaged, even on hardware that is extremely constrained in memory. In the following chapter, we therefore present several concrete applications that take advantage of OMicroB and OCaLustre.

8 Applications for Electronic Devices

We now present a set of practical or recreational applications developed with the software solutions described in this dissertation. These applications, which can be executed on microcontrollers with very limited memory and computational resources, highlight the value of the high-level approach adopted in this thesis. Indeed, it becomes straightforward to implement small embedded programs that take advantage of both the guarantees provided by the OCaml language and the synchronous layer of OCaLustre.

We illustrate this with three example programs, chosen to demonstrate the expressiveness of our solution while validating its compatibility with hardware constrained to only a few kilobytes of memory. In particular, the programs described below can be executed on devices with less than 8 kilobytes of RAM.

Each program presented is tied to a concrete electronic setup. The first example is a program that controls a simple punched-card reader, where each card encodes the binary representation of one byte. This example highlights the correspondence between the notion of a synchronous *clock* and that of a physical clock, which governs the timing of electrical signals arriving at the program inputs. The second example manages the operation of a *chocolate tempering machine*a kitchen appliance designed to melt chocolate to a precise temperature. The final example is the implementation of a simple *Snake* video game for the *Arduboy*, a small handheld device dedicated to running simple games, typically programmed in C.

The complete source code for these programs is provided in Appendix D.

8.1 A Punched-Card Reader

For our first example, we develop a microcontroller program integrated into a setup simulating a punched-card reader. Each card encodes the representation of a single byte using two horizontal rows of *holes*. The first row carries the *clock* signal : whenever the circuit detects a hole in this row, it means that a data value must also be read. This *data* is represented on the second row of the card : a hole corresponds to the binary value 1, while the absence of a hole corresponds to the value 0.

Figure 8.1 illustrates the design of such a punched card. The depicted card encodes the byte 0*b*10001010[1].

8.1.1 Circuit

A fairly simple electronic circuit, shown in Figure 8.2, can be used to build a reader for such punched cards. Two digital input pins (I_0 and I_1 in the figure) of a microcontroller are connected to two metal rods, each applying light pressure on one of the horizontal rows of holes in the card. The card itself rests on a metal plate supplied with an electrical current.

When the punched card is slid between the plate and the metal rods, the values represented by the presence or absence of holes can be detected. If a rod passes over a hole, an electrical contact is

1. The byte is read from left to right, using a *big-endian* representation.

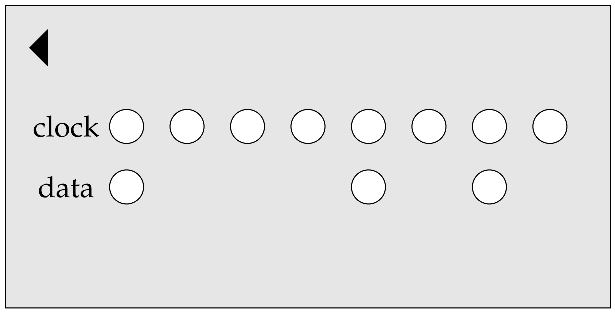


FIGURE 8.1 – Schematic representation of a punched card

established between the rod and the plate, and the corresponding microcontroller pin is connected to the power source : the value 1 (*HIGH*) is registered. Conversely, when a metal rod rests on the card, no contact is made (since the card is non-conductive), and the microcontroller pin is instead connected to ground via a *pull-down* resistor : in this case, the value 0 (*LOW*) is measured.

Finally, the binary value read from the card is output by the microcontroller to a set of eight LEDs (connected to pins O_0 through O_7). Each LED represents one bit of the byte encoded on the card : the LED lights up if the bit is 1, and remains off if the bit is 0.

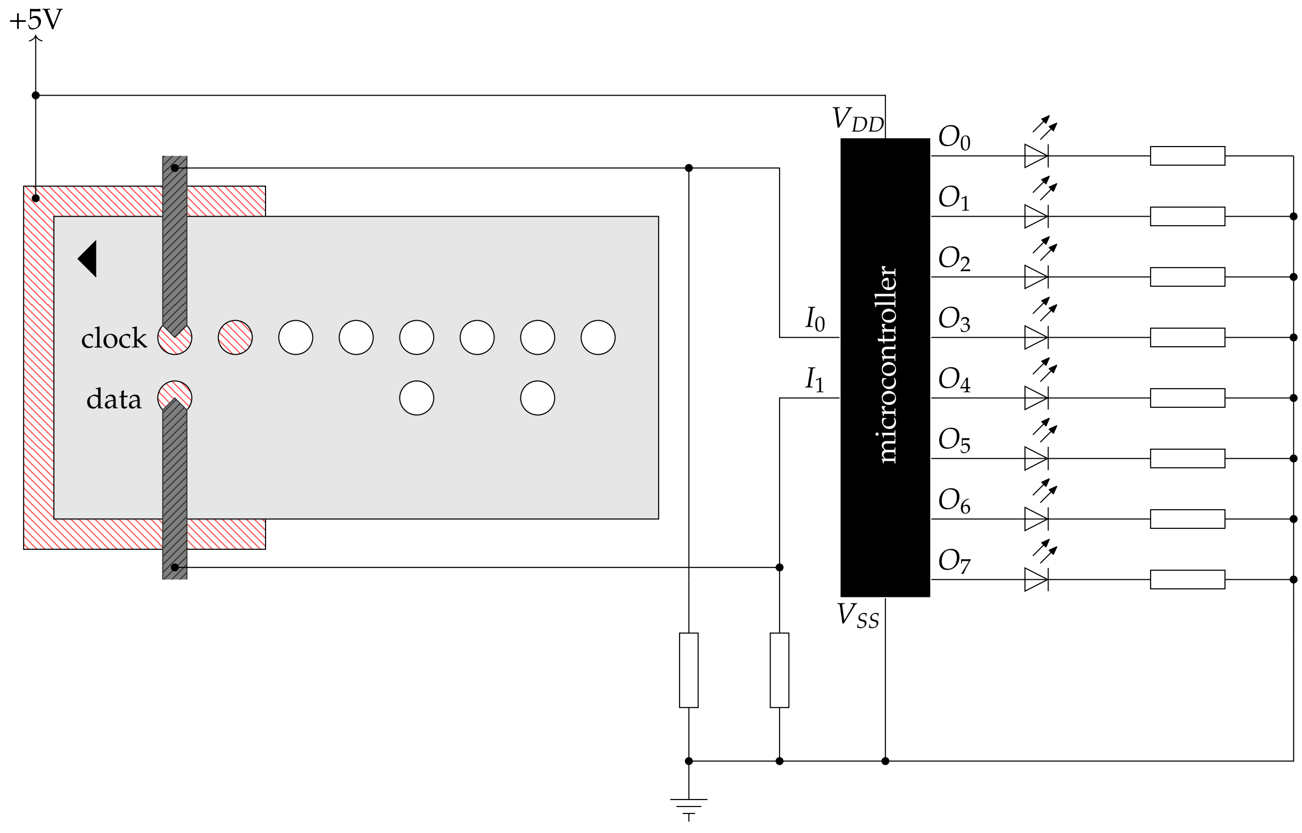


FIGURE 8.2 – Circuit for a punched card reader

8.1.2 Program

The use of a clock signal physically represented by a sequence of holes on the punched card illustrates well the notion of a clock in a synchronous program. An OCaLustre program can therefore be easily written to control the circuit described above. In this program, a clock signal must be set to *true* whenever the metal pin positioned on the line corresponding to the clock signal falls into a hole. However, as long as the pin remains in the same hole, a continuous stream of `true` values should not be produced : it is in fact the *entry* into a hole that must indicate a *tick* of the clock signal. To achieve this, we define a node `edge` that detects such a rising edge :

```
let%node edge x ~return:e =
  e = (x && (not (true ⋙ x)))
```

Using this node, we then define the node `read_bit`, which reads a bit on the card whenever the clock signal is true :

```
let%node read_bit (top,bot) ~return:(clk,data) =
  clk = edge top;
  data = bot [@when clk]
```

The flow `top` corresponds to the state of the first line of the punched card, and the flow `bot` to the state of the second line. It is worth noting here that the « physical » representation of the clock by holes in the punched card is directly reflected in the definition of the synchronous clock signal `clk`, which represents the sampling frequency of the data signal `data`.

The program must detect when eight bits have been read. To achieve this, we declare a node `count` that counts the number of synchronous instants, from 0 up to the integer preceding a reset value named `reset` :

```
let%node count (reset) ~return:(c) =
  c = (0 ⋙ (c + 1)) mod reset
```

Each time the clock signal is true, the value of the data signal is stored in an array of 8 cells. Such an array `t` is declared in OCaml as follows :

```
let t = Array.make 8 false
```

In this program, once eight bits have been read, the corresponding byte will be displayed on eight LEDs connected to the microcontroller. We then define a main node `read_card`, which represents the complete reading of a punched card : once eight bits have been read, the flow `send` is set to true, signaling to the OCaml program that it can update the state of the LEDs :

```
let%node read_card (top,bot) ~return:(i,data,clk,send) =
  (clk,data) = read_bit (top,bot);
  i = count(8 [@when clk]);
  send = merge clk (i = 7) false
```

The index of the array cell corresponding to the bit currently being read is represented by the flow `i`. This flow is only present (and its value incremented) when the clock `clk` is true.

On an AVR ATmega2560 microcontroller, where the metal rods are connected to pins 13 and 12, and the LEDs are connected to pins 42 to 49, the OCaml code responsible for the program's input/output is as follows :

```
open Avr

let t = Array.make 8 false
let clk = PIN13
let data = PIN12
let leds = [| PIN42 ; PIN43; PIN44; PIN45; PIN46; PIN47; PIN48; PIN49 |]

let init () =
  (* configure pins as input *)
  pin_mode clk INPUT;
  pin_mode data INPUT;
  (* configure pins as output *)
  Array.iter (fun x -> pin_mode x OUTPUT) leds

let input_clk () = bool_of_level (digital_read clk)
let input_data () = bool_of_level (digital_read data)

(* function that turns an LED on or off *)
let update_led i b = digital_write leds.(i) (level_of_bool b)

let output i data clk send =
  if clk then t.(i) ← data;
  if send then Array.iteri update_led t
```

Finally, the code of the program's main loop, which provides the interface between these functions and the synchronous OCaLustre program, is shown below :

```
let () =
 (* hardware initialization *)
 init ();
 (* creation of the main node state *)
 let st = read_card_alloc () in
 while true do
   (* reading inputs *)
   let c = input_clk () in
   let d = input_data () in
   (* updating the state of the main node *)
   read_card_step st c d;
   (* emitting outputs *)
   let i = st.read_card_out_i in
   let data = st.read_card_out_data in
   let clk = st.read_card_out_clk in
   let send = st.read_card_out_send in
   output i data clk send
 done
```

This code is compatible with the OCaLustre compilation model designed for real-time embedded systems. Thus, during execution of the program's main loop, no new OCaml values are allocated on the heap : the *garbage collector* is therefore never triggered during execution of the synchronous program, and the execution time of a synchronous instant can then be measured using *Bytecrawler*. The `-m` option, followed by the name of the main node, automatically generates the code of the program's main loop, as well as the prototypes of the functions that allow interaction between the synchronous program and its environment. We will illustrate the use of this option in the following examples.

8.1.3 Static analyses of the program

Using the tools described earlier in this manuscript, we can perform several static analyses on the program's source code, as well as on the associated bytecode.

Clock typing : The clock types induced by the clock inference algorithm for the nodes of the punched card reader program are as follows[2] :

```
edge :: (x:base) -> (e:base)
read_bit :: (top:base * bot:base) -> (clk:base * data:(base on clk))
count :: (reset:base) -> (c:base)
read_card :: (top:base * bot:base) -> (i:(base on clk) * data:(base on clk) * clk:base * send:base)
```

These types are exactly as expected : in particular, the values of the flows `i` and `data` are present only when the clock flow `clk` is true. The use of the `merge` operator to define the flow `send` makes it possible to oversample the value `(c=7)` on the base clock. This clock typing information must be taken into account when implementing the interface functions between the synchronous program and the OCaml program : for example, it would be incorrect for these functions to access the value of the flow `i` when the flow `clk` is false.

2. Typing information for the nodes can be displayed using the `-i` option of OCaLustre.

Worst-case execution time : On an AVR ATmega2560 microcontroller, the worst-case execution time of a synchronous instant of this program, which includes calls to input/output primitives, is estimated by the Bytecrawler tool at 57,162 cycles.

It should be noted, however, that the use of arrays in the program involves calls to specific C primitives for reading and writing array cells. By default, these primitives check that the indices of the accessed cells are strictly less than the array size, and raise an exception otherwise. However, the Bound-T tool (which we also use to estimate the execution time of the primitives) cannot determine the execution time of a function that may raise an exception. Therefore, we compile the program with the `-unsafe` option of the *ocamlc* compiler, in order to disable these array bound checks, thus making it possible to estimate the maximum execution times of the primitives that perform array reads or writes.

Since the microcontroller used has a clock frequency of 16 MHz, our punched card reader is able to process a pair (*clock*, *data*) in about 3.57 milliseconds. This short duration ensures that the user of the punched card cannot, in practice, slide a card too quickly through the device and cause a faulty read.

8.1.4 Memory consumption

The minimum stack size required for this program to run smoothly is 36 values, while the minimum heap size is 318 values. Thus, in a 16-bit configuration of the virtual machine, only 879 bytes of RAM are needed for this program to execute. Its flash memory footprint is about 12.1 kilobytes, which makes it compatible with microcontrollers with even fewer memory resources than the ATmega2560, such as the ATTiny1614, which has 16 kilobytes of flash memory and 2048 bytes of RAM, and whose purchase cost is around 60 euro cents.

Enabling early evaluation, which consists of precomputing the values from the initialization of OCaml modules at compile time, further reduces memory usage : the program then requires only a stack of 35 values and a heap of 160 values, for a flash memory footprint of 10.8 kilobytes and a RAM usage of just 533 bytes.

8.2 A Chocolate Tempering Machine

For our second application example, we describe the circuit and program for implementing a chocolate tempering machine. This device is a kitchen appliance that heats chocolate to a temperature specified by the user. Our tempering machine is built from a relatively inexpensive electronic setup. In this circuit, an ATmega2560 microcontroller is connected to the following electronic components :

— An electric heating resistor that converts an electrical signal into heat.
— A temperature sensor that converts the temperature of its environment into an analog value.
— A two-segment liquid crystal display (LCD) of type LCD1602A, often provided in microcontroller development kits.
— Two push buttons.

Figure 8.3 is a schematic representation of this electronic circuit.

The components of this circuit are used as follows : the heating resistor is placed in a container filled with water, and the chocolate is then heated in a bain-marie by the tempering machine, which controls the temperature of the water. Whenever the current temperature of the preparation is below the target temperature set by the user, the heating resistor is activated to raise the temperature. The interface

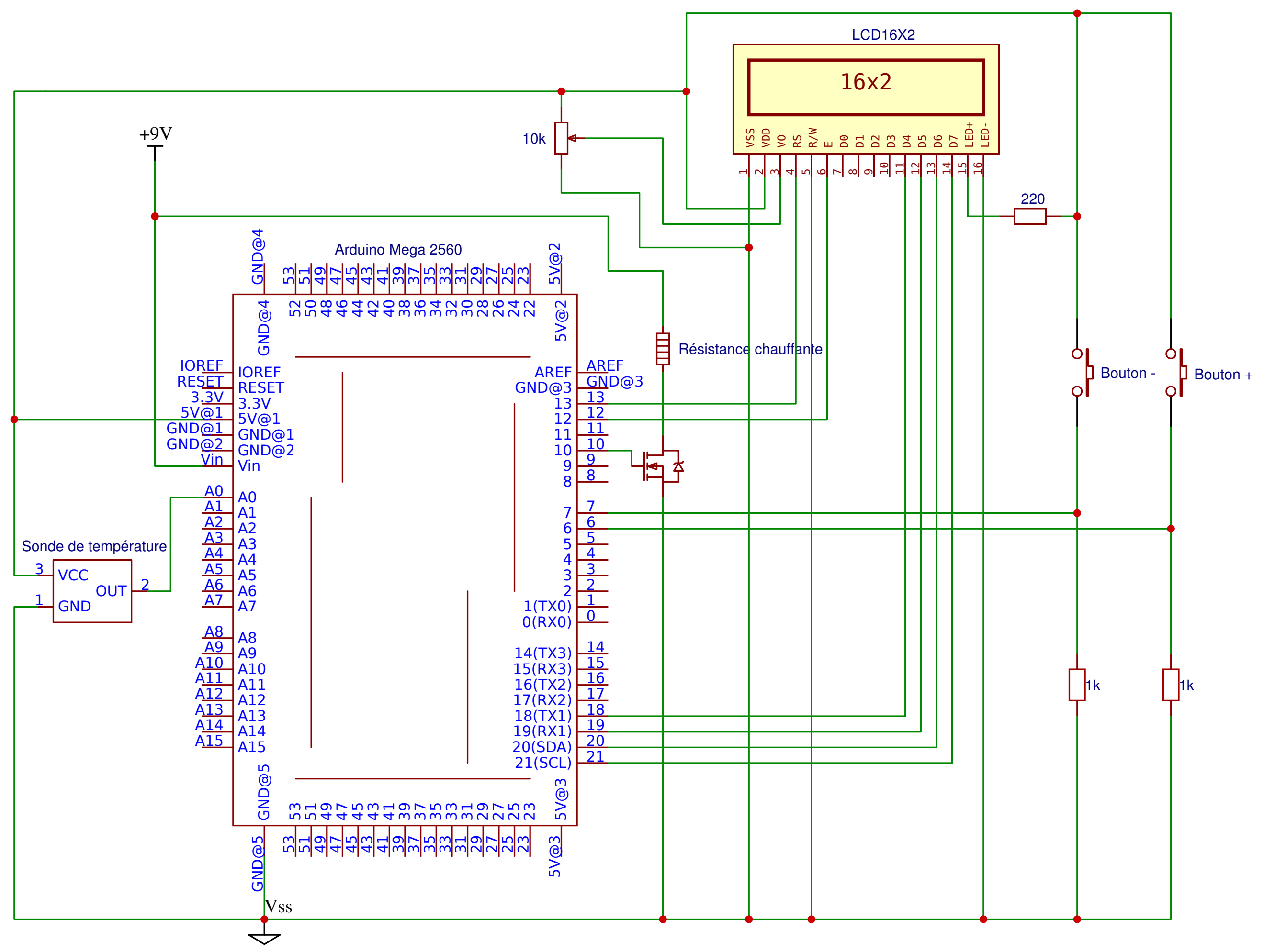


Figure 8.3 – Schematic representation of the chocolate tempering machine

consisting of the LCD screen and the two push buttons allows the user to increase or decrease the desired temperature.

The role of the program associated with this circuit is therefore to send an electrical signal to the pin connected to the resistor depending on the value returned by the temperature sensor. The desired temperature is calculated based on successive presses of the push buttons. In parallel, the program continuously displays both the current temperature of the preparation and the target temperature on the LCD screen.

8.2.1 OCaLustre Program

The microcontroller at the core of this circuit is responsible for executing the program by *synchronizing* the different electronic components involved in the operation of the tempering machine. The synchronous programming model is therefore particularly well suited to such an embedded program : each node of the synchronous program represents a component of the application, which must execute at the *same time* as the others, and their interaction governs the operation of the complete device.

The synchronous program associated with this circuit is then very simple : only three OCaLustre nodes are required to define the behavior of the program :

```
(** turn on/off if both + and - are pressed simultaneously **)
let%node thermo_on (p,m) ~return:(b) =
  b = (true ⋙ if p && m then not b else b)

(** modify the desired temperature depending on which button is pressed **)
let%node set_wanted_temp (p,m) ~return:(w) =
  w = (325 ⋙ if p then w+5 else if m then w-5 else w)

(** main node: computes desired temperature and the state of the heater **)
(** Temperatures are expressed in tenths of degrees Celsius **)
let%node thermo (plus,minus,real_temp) ~return:(on,wanted,real,resistor) =
  on = thermo_on (plus,minus);
  wanted = set_wanted_temp (plus[@when on], minus[@when on]);
  real = real_temp [@when on];
  heat = (real < wanted);
  resistor = merge on heat false
```

The program manipulates temperature values as integers representing tenths of degrees Celsius. In particular, the main node receives the values of the two push buttons and the measurement from the temperature sensor, and produces five distinct flows :

— The flow on distinguishes the active state of the device from the standby state : the tempering machine switches between the two whenever both push buttons are pressed at the same time. This flow is the clock that governs the presence of most flows handled by the program.

— The flow `wanted` is defined as the result of the call to the node `set_wanted_temp`, which allows the desired temperature (initialized at 32.5 °C) to be increased or decreased in increments of 0.5 °C, depending on the buttons pressed by the user of the tempering machine. This flow is only defined when the on flow is true.

— The flow `real` is the result of sub-sampling the flow `real_temp`, which represents the value measured by the temperature sensor, using the clock `on`.
— The flow `heat` defines the condition under which the heating resistor should be turned on : if the actual temperature is lower than the desired temperature, then this flow is *true*. It should be noted that, in order to keep the examples in this chapter simple, this version of the tempering machine is extremely naïve : the inertia of the heating resistor can cause the temperature to rise above the desired value. A real device would take into account various physical aspects of the system's environment in order to control temperature variations more preciselyfor example, by stopping the heating as soon as the preparation reaches a temperature close to the desired one, or by calculating the heating duty cycle required to maintain a stable temperature. However, such refinements go beyond the purpose of this chapter, which is to illustrate, through simple examples, possible applications of synchronous programming with OCaLustre. An OCaLustre program that computes the heating duty cycle is nevertheless provided in Appendix D.2.3.
— The flow `resistor` results from merging the previous flow with a constant flow computing the value `false`. This flow indicates whether the heating resistor should be powered. When the device is off, the resistor must be off (`false`); when the tempering machine is on, the value of the `heat` flow determines the state of the heating resistor.

Clock typing and node signatures : The clock types of the nodes automatically inferred by OCaLustre during compilation are as follows :

```
thermo_on :: (p:base * m:base) -> (b:base)
set_wanted_temp :: (p:base * m:base) -> (w:base)
thermo :: (plus:base * minus:base * real_temp:base)
              -> (on:base * wanted:(base on on) * real:(base on on) * resistor:base)
```

In accordance with the description of the different flows defined by the program, the flows `wanted` and `real` returned by the node `thermo` are indeed clocked by the flow `on` : their values are therefore only available when the tempering machine is active.

8.2.2 Interaction loop

The `-m <node>` option of OCaLustre generates the execution engine code of the program, responsible for repeatedly calling the node `<node>`. This node then acts as the main node. This option generates a file `<node>_io.ml` pre-filled with the signatures of the functions `init_<node>`, `input_<node>`, and `output_<node>`, which correspond to the interaction functions between the synchronous kernel of the program and the rest of the OCaml code.

In addition, the `-d <ms>` option allows the main loop to be clocked so that one execution of a synchronous instant of the program is performed every `<ms>` milliseconds. Since the chocolate tempering program does not involve strict timing constraints, an execution frequency of 1/500 ms seems sufficient : the program is therefore compiled with the options `-m thermo -d 500`.

An `thermo_io.ml` file containing the prototypes of the functions that enable interaction between the synchronous part of the program and its environment is then generated by OCaLustre. Once completed, the code of these functions is as follows :

```
(*** Interface functions for the synchronous instant ***)

(** initialization function **)
let init_thermo () =
  (* initialize analog reading *)
  Avr.adc_init ();
  (* initialize the display *)
  LiquidCrystal.lcdBegin lcd 16 2;
  (* initialize the pins *)
  pin_mode sensor INPUT;
  pin_mode resistor OUTPUT;
  pin_mode plus INPUT;
  pin_mode minus INPUT

(** input function **)
let input_thermo () =
  let plus = digital_read plus in
  let minus = digital_read minus in
  let plus = bool_of_level plus in
  let minus = bool_of_level minus in
  let real_temp = read_temp () in
  (plus,minus,real_temp)

(** output function **)
let output_thermo (on,wanted,real,res) =
  if on then
    begin
      print_temp wanted real;
      digital_write resistor (if res then HIGH else LOW)
    end
  else idle ()
```

These functions make use of auxiliary OCaml functions that are mainly responsible for converting the temperature measured by the electronic sensor into degrees Celsius, as well as displaying them on the LCD screen. The code for these functions, along with the declarations of the different variables used by the program, is as follows :

```
open Avr

(* declaration of the LCD screen *)
let lcd = LiquidCrystal.create4bitmode PIN13 PIN12 PIN18 PIN19 PIN20 PIN21

(* declaration of pins *)
let plus = PIN7
let minus = PIN6
let resistor = PIN10
let sensor = PINA0

(* temperature conversion *)
let convert_temp t =
  let f = (float_of_int (1033 - t) /. 11.67) in
  int_of_float (f*.100.)

(* temperature reading *)
let read_temp () =
  let t = analog_read sensor in
  convert_temp t

(* displaying the temperatures on the LCD screen *)
let print_temp wanted real =
  let split_temp t =
    let u = t/10 in
    let dec = t mod 10 in
    (u,dec) in
  LiquidCrystal.home lcd;
  let (wu,wd) = split_temp wanted in
  let (ru,rd) = split_temp real in
  LiquidCrystal.print lcd "Wanted T :";
  LiquidCrystal.print lcd ((string_of_int wu)^"."^(string_of_int wd));
  LiquidCrystal.setCursor lcd 0 1;
  LiquidCrystal.print lcd "Actual T :";
  LiquidCrystal.print lcd ((string_of_int ru)^"."^(string_of_int rd))

let idle () =
  LiquidCrystal.home lcd; LiquidCrystal.clear lcd; LiquidCrystal.print lcd "..."
```

8.2.3 Memory Consumption

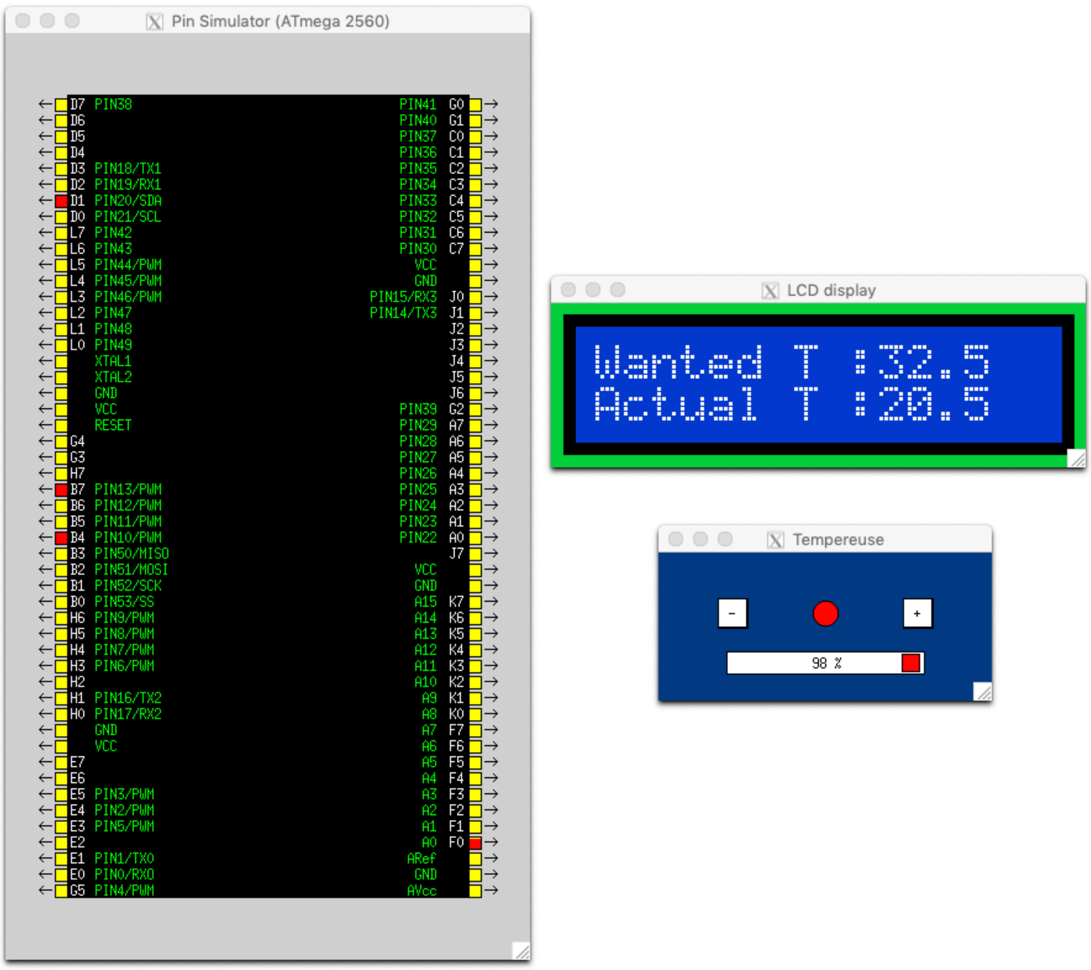
The chocolate tempering program requires at minimum a stack of 61 values and a heap of 448 values, which corresponds, in a 16-bit representation of OCaml values, to a total memory footprint of 17.3 kilobytes of flash memory and only 1277 bytes of RAM (with the Stop and Copy GC[3]). Nevertheless,

3. It should be noted that due to the use of this GC algorithm, the number of OCaml values that can actually be allocated corresponds to half the heap size.

since the frequent activation of the GC algorithm slows down program execution (as illustrated in the previous chapter), it would be unfortunate not to take advantage of the full extent of the RAM of the microcontroller being used. A heap of 3800 words thus makes it possible to fill nearly all 8 kilobytes of RAM of the ATMega2560 microcontroller, reducing the number of GC triggers during the first one hundred synchronous instants from 699 to just 4.

8.2.4 Simulation

The simulator integrated into OMicroB makes it possible to run this program directly on a computer, mainly to facilitate debugging. Figure 8.4 shows the simulator interface for this program : the state of the heating resistor is represented here by an LED located between the two buttons + and –, which allow the user to adjust the temperature. The horizontal bar located below these elements is used to set the value measured on the analog input pin to which the temperature sensor is actually connected.



```
Wanted T :32.5
Actual T :20.5
```

FIGURE 8.4 – Execution of the chocolate tempering program in the simulator

8.3 A Snake Game

The final application presented in this dissertation is a small video game of « Snake », in which a snake, represented by a sequence of contiguous cells, moves in a two-dimensional world to eat apples that appear as the game progresses. Each time the snake eats an apple, it grows by one cell, and a new apple appears on the screen. The objective of the game is for the snake to eat as many apples as possible in order to reach a certain size, while avoiding collisions with itself, which would kill it (and cause the player to lose).

This video game was very popular in the late 1990s and early 2000s, during the rise of mobile phones : phones of that era, whose computing speeds were comparable to those of the microcontrollers studied in this dissertation[4], could run a handful of simple games adapted to the limited performance of mobile processors of the time. For this reason, we detail in this section an implementation of this game adapted to a small device with very limited hardware resources. This device, called the *Arduboy*, is a small portable game console the size of a credit card, designed for the development of small games, often inspired by the *retrogaming* community. For performance considerations and precise use of the device's resources, programs developed by the Arduboy community are generally written in C. However, through this example, we show that it is entirely feasible to develop an application on it using the OMicroB/OCaLustre pair. The software solutions presented in this dissertation thus enable the use of high-level abstractions on hardware with very restricted memory resources. Figure 8.5 shows an Arduboy running the Snake game.

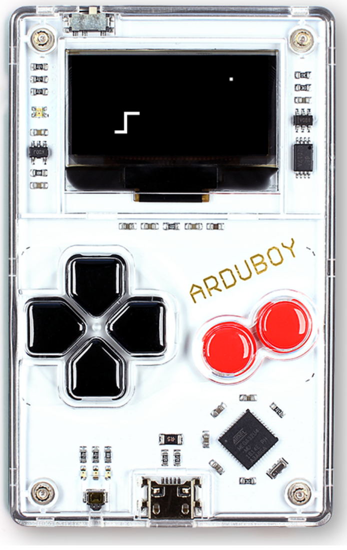


FIGURE 8.5 – An Arduboy

The complete source code of the modules presented in this section is available in Appendix D.3.

8.3.1 Anatomy of an Arduboy

The main component of an Arduboy is a microcontroller from the AVR family : the ATmega32u4. This microcontroller falls into what we consider to be highly resource-constrained hardware : while its theoretical clock speed can reach 16 MHz, the Arduboy's battery has a fairly low voltage (between 3.4V and 3.7V), so its actual performance is closer to 1112 MHz. Furthermore, the ATmega32u4 has 32 KB of flash memory and only 2.5 KB of RAM, which requires programs to have a very small memory footprint.

An Arduboy also contains several electronic components, mostly for user interaction : it has six push buttons (four directional arrows up, down, left, and right and two action buttons A and B), a

4. For example, the Nokia 3310, released in 2000, had a Texas Instruments MAD2WD1 processor running at 13 MHz.

monochrome OLED (*Organic Light-Emitting Diode*) display with a resolution of 128 × 64 pixels, three LEDs (one red, one green, and one blue) located next to the screen, as well as a piezoelectric speaker capable of playing low-definition sounds. For our application, we will use the display to show the snake and the apple to be eaten, the left and right direction buttons to move the snake, and the LEDs to indicate whether the game is won (green) or lost (red).

The circuit diagram representing the Arduboy components used by our program is shown in Figure 8.6.

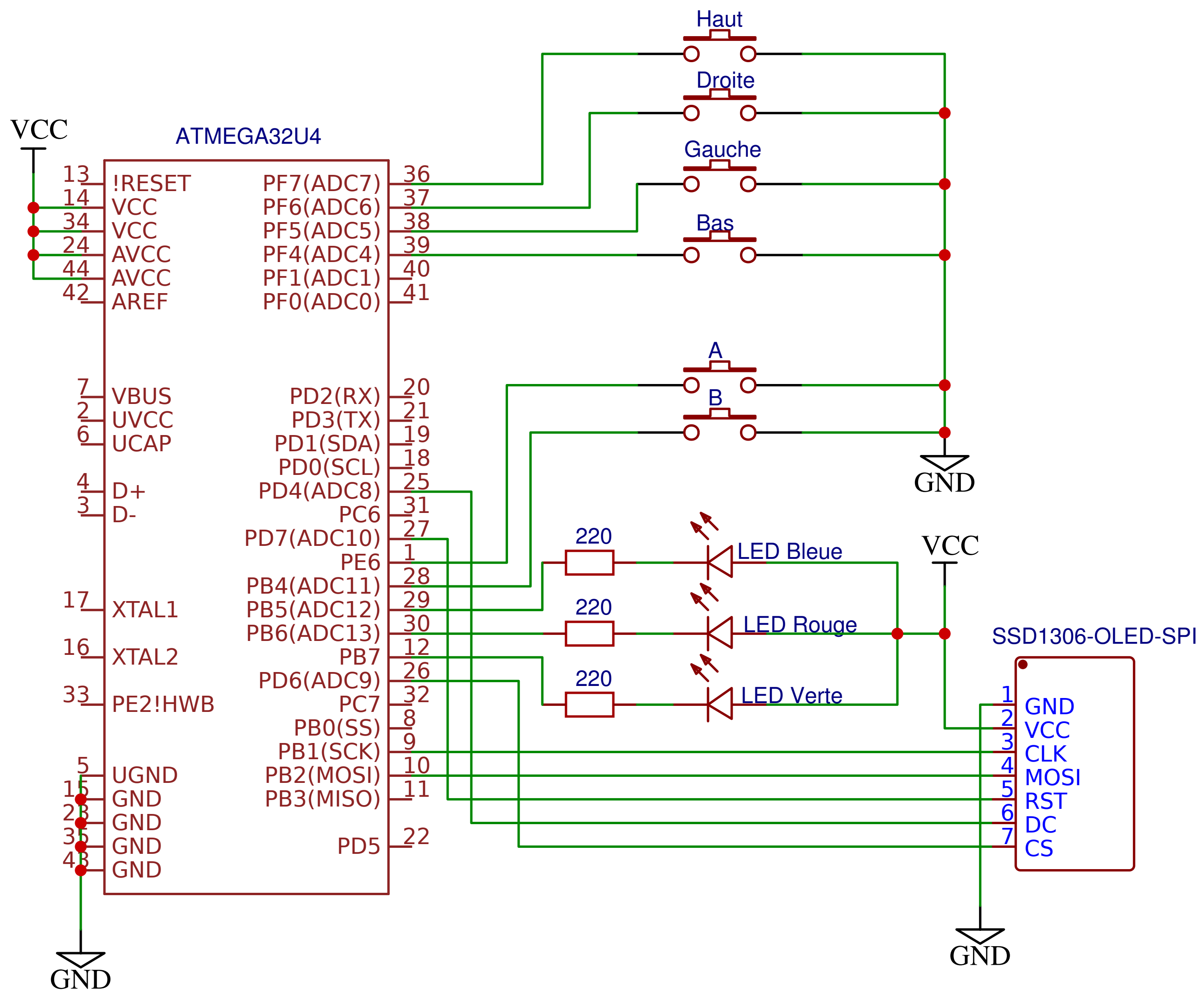


FIGURE 8.6 – Arduboy components used by the Snake game

8.3.2 Program for the Electronic Circuit

The program takes into account the technical specificities of the Arduboy : we first describe the OCaml modules that allow interaction between the microcontroller and the input/output components, which are necessary for the proper functioning of the game.

SPI Connection and OLED Display Interface : On an Arduboy, data transfer between the microcontroller and the OLED display (SSD 1309 type) is carried out via an SPI (*Serial Peripheral Interface*) connection that uses two wires :

— One wire is connected to the SCK (*Serial Clock*) pin of the microcontroller and carries the clock signal that synchronizes the communication.

— One wire is connected to the MOSI (*Master Output, Slave Input*) pin of the microcontroller and carries the command or data signals sent to the display.

The SPI connection is implemented directly in hardware by the microcontroller : it only needs to be properly initialized[5], after which any byte placed in the SPDR register is transmitted by the microcontroller. The OCaml code that configures and interacts with the SPI connection is provided in a module named *Spi*, whose implementation is shown in Figure 8.7. This module makes use of the *Avr* module provided in the OMicroB standard library, which defines the registers and pins of the microcontroller, along with functions to modify their state.

```
(*** Module for managing the SPI (Serial Peripheral Interface) connection ***)

open Avr

(** Initialize the SPI connection **)
let begin_spi ~sck ~mosi =
  set_bit SPCR MSTR;
  set_bit SPCR SPE;
  set_bit SPSR SPI2x;
  pin_mode sck OUTPUT;
  pin_mode mosi OUTPUT

(** Stop the SPI connection **)
let end_spi () = clear_bit SPCR SPE

(*** Transmit data via the SPI connection ***)
let transfer data = write_register SPDR data
```

FIGURE 8.7 – `spi.ml` module for interacting with a serial peripheral

Moreover, the signals used to control the display are carried by three separate wires :

— A wire connected to the D/C (*Data/Command*) pin of the display sets it to either « command » mode or « data » mode.

— A wire connected to the CS (*Chip Select*) pin of the display enables communication between the display and the microcontroller only when it is at a low level (*LOW*).

— A wire connected to the RST (*Reset*) pin of the display resets the microcontroller : if this pin goes from high (*HIGH*) to low (*LOW*), the display resets.

In addition, to illustrate interoperability between OCaml and the C language, the display data will be stored in a C array of bytes (in which each byte represents 8 points on the screen) acting as a buffer. It should be noted that, in order to reduce the memory footprint of the program and simplify the readability of the game, the screen resolution actually used by Snake will be 64×32 points. The display

5. The technical details, which involve configuring the SPI connection by setting specific bits in several registers, are outside the scope of this dissertation.

will nevertheless take up the entire width and height of the screen : the game's « points » will therefore appear as 2×2 pixels on the display.

The OCaml module *Oled*, an excerpt of which is shown in Figure 8.8, contains the functions required for handling the display and managing the buffer. In particular, a function `Oled.flush` allows transferring a new image representing the current state of the game to the display. Details about the display's characteristics and configuration are available in the SSD1306 datasheet [⚓4].

```
(*** Draw a point (true=black / false=white) ***)
let draw x y color = write_buffer x y color

(*** Clear the screen ***)
let clear () =
 for _i = 0 to 1023 do
   Spi.transfer(0x00)
 done

(*** Send the buffer to the display ***)
let flush () =
  for _i = 0 to 1023 do
    Spi.transfer(get_byte_buffer())
  done

(*** Initialize the display ***)
let boot ~cs ~dc ~rst =
  digital_write rst HIGH;
  digital_write rst LOW;
  digital_write rst HIGH;
  command_mode cs dc;
  transfer_program boot_program;
  data_mode cs dc;
  clear()
```

FIGURE 8.8 – `oled.ml` module (excerpt) for configuring and controlling the SSD1306 display

Arduboy Configuration : Finally, an Arduboy module (fig. 8.9) configures the device's components using functions from the standard library. The pin numbers follow the Arduino naming convention (in this case, that of the Arduino Leonardo, which also embeds an ATMega32u4 microcontroller).

— The function `pin_mode` allows one to configure a pin as an input (`INPUT`), an output (`OUTPUT`), or an input connected through a *pull-up* resistor. AVR pins include an internal *pull-up* resistor, which has the effect that the value read on a pin is high (*HIGH*) when it is not connected to any component. Consequently, when the Arduboy's push buttons are open, the value read is *HIGH*, and when they are closed, the value read is *LOW* because they are connected to the circuit ground.

— The function `digital_write` allows one to write a binary value (high level or low level) to a pin.

This module, whose implementation follows the wiring diagram that represents the connections between the microcontroller and the external components, configures all the hardware required for this application : in particular, the `init` function configures the pins used by the program, activates the SPI interface, and initializes the display.

```
open Avr

(** Pins used **)
let cs = PIN12
let dc = PIN4
let rst = PIN6
let button_left = PINA2
let button_right = PINA1
let button_down = PINA3
let button_up = PINA0
let button_a = PIN7
let button_b = PIN8
let blue = PIN9
let red = PIN10
let green = PIN11

(** Initialize RGB LEDs with common anode (HIGH = off) **)
let init_led l =
  digital_write l HIGH

(** Turn on one of the RGB LEDs with common anode (LOW = on) **)
let light_led l =
  digital_write l LOW

(** Initialize pins **)
let boot_pins () =
  List.iter (fun x -> pin_mode x INPUT_PULLUP) [button_left; button_right; button_up;
                                                 button_down];
  pin_mode button_a INPUT_PULLUP;
  pin_mode button_b INPUT_PULLUP;
  List.iter (fun x -> pin_mode x OUTPUT) [red;green;blue];
  List.iter (fun x -> pin_mode x OUTPUT) [cs;dc;rst];
  List.iter init_led [red;green;blue]

(** Initialize pins, SPI link, and the display **)
let init () =
  boot_pins ();
  Spi.begin_spi ~sck:SCK ~mosi:MOSI;
  Oled.boot ~cs:cs ~dc:dc ~rst:rst
```

FIGURE 8.9 – `Arduboy.ml` module : configuration and initialization of hardware

The different modules presented in this subsection together implement a *library* for controlling the internal hardware of the Arduboy. This library can be used for the implementation of our Snake game, but it can also be reused in many other projects targeting this hardware. Indeed, all the functionality presented in this section simply reflects the characteristics of the electronic circuit and the relationships between its various components, which do not change from one program to another. This library illustrates the fact that using a high-level language such as OCaml is also well suited for defining low-level interactions, as is the case here with the electronic interactions between the ATmega32u4 microcontroller and the components it is connected to.

8.3.3 Game Program

Building on the library described in the previous section, we now turn to the details of the Snake game implementation. This game has the structure of a synchronous program that uses both an OCaLustre kernel to represent the reaction to player actions, and OCaml functions whose main role is to handle communication between this kernel and external peripherals such as the display or the directional buttons.

In this program, the snake is represented by an array `snake` in which each cell stores the positions of the various sections of the snake's body :

```
let max_size = 15
let snake = Array.make max_size (0,0)
```

Two values, `head` and `tail`, represent the indices of the array cells corresponding respectively to the head and the tail of the snake. To represent the movement of the snake, at each instant of the program these two pointers each advance to the next higher index, and the position of the new head of the snake is written into the cell now pointed to by `head`. As illustrated in figure 8.10, the array has a circular structure : whenever either of these pointers moves past the end of the array, it is reset to index 0.

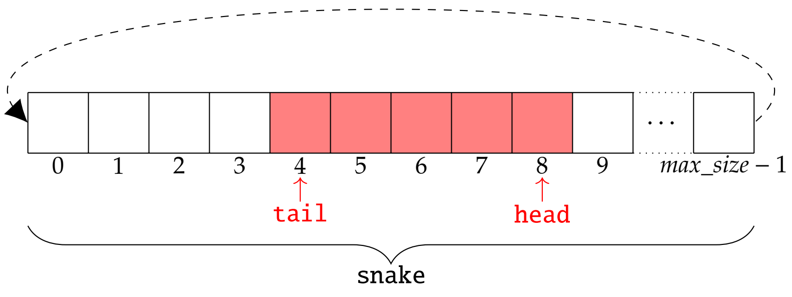


FIGURE 8.10 – Structure of the snake

Synchronous kernel : The game loop is responsible for computing the direction of the snake based on whether the right or left button is pressed, calculating the new position of its head, and also taking into account the fact that the snake grows whenever its head reaches the same position as the apple (in other words : whenever it eats the apple). The code for this loop is an OCaLustre node that takes as input the maximum size of the snake, the state of the left and right buttons, as well as the dimensions of the world. It computes the new values of the `head` and `tail` pointers, the new position of the snake's head (`new_x,new_y`), the position of the apple (`apple_x,apple_y`), and indicates whether the player has won (`win`) :

```
    ~return:(head,tail,new_x,new_y,apple_x,apple_y,win) =
  (new_x,new_y) = new_head (dir,width,height);
  dir = direction(left,right);
  head = (1 >>> ((head+1) mod max_size));
  eats = (apple_x = new_x && apple_y = new_y);
  (a_x,a_y) = new_apple (width [@when eats], height [@when eats]);
  apple_x = (10 >>> merge eats a_x (apple_x [@whennot eats]));
  apple_y = (10 >>> merge eats a_y (apple_y [@whennot eats]));
  tail = merge eats ((0>>> tail)[@when eats]) (0 >>> ((tail+1) mod max_size)[@whennot eats]);
  size = (1 >>> merge eats ((size + 1) [@when eats]) (size [@whennot eats]));
  win = (size = max_size - 1)
```

This node defines a clock `eats`, which is true whenever the snake eats the apple. This clock governs the execution of several computations :

— When `eats` is true, a new position (`a_x,a_y`) for the apple is computed by a node `new_apple` :

```
(** Random draw of an integer **)
let new_position n = Random.int n

(** New position of the apple **)
let%node new_apple (width,height) ~return:(a_x,a_y) =
  (a_x,a_y) = (call new_position width, call new_position height)
```

— When `eats` is true, the snake's size `size` increases.

— If `eats` is true, the pointer to the tail of the snake (`tail`) is not shifted, which simulates the snake growing longer.

The game loop also calls a node `direction` to compute the direction of the snake :

```
let left_of = function South -> East | North -> West | East -> North | West -> South
let right_of = function South -> West | North -> East | East -> South | West -> North

(** Turn left **)
let%node left (dir) ~return:ndir =
  ndir = call left_of dir

(** Turn right **)
let%node right (dir) ~return:ndir =
  ndir = call right_of dir

(** Snake's direction based on its previous direction **)
let%node direction (l,r) ~return:dir =
  pre_dir = (South ⋙ dir);
  dir = if l then
          (merge l (left (pre_dir [@when l])) (pre_dir [@whennot l]))
        else
          (merge r (right (pre_dir [@when r])) (pre_dir [@whennot r]))
```

In particular, the use of the operator ⋙ makes it possible to define the flow `dir` based on its previous value : thus, at the beginning of the program the snake moves south, then, when the left button (resp. right) is pressed, it turns left (resp. right) relative to its previous position. The snake keeps the same direction if no button is pressed. The node `game_loop` also calls a node `new_head`, which computes the new position of the head :

```
(** Modulo **)
let nmod x y = (x + y) mod y

(** New coordinate **)
let%node new_coord (dir,max,v,dir1,dir2) ~return:n =
  n = if dir = dir1 then call nmod (v-1) max
      else if dir = dir2 then call nmod (v+1) max
      else v

(** New position of the snake's head **)
let%node new_head (dir,w,h) ~return:(x,y) =
  x = new_coord (dir,w,0 ⋙ x,West,East);
  y = new_coord (dir,h,0 ⋙ y,North,South)
```

Finally, the main node `main` of the OCaLustre program is responsible for detecting a rising edge on the two buttons that allow the snake to turn left or right, and calls the node implementing the game loop :

```
(** Rising edge (for the buttons) **)
let%node rising_edge i ~return:o =
  o = (i && (not (false ⋙ i)))

(** Main node **)
let%node main (max_size,button1,button2,width,height)
    ~return:(head,tail,nx,ny,apple_x,apple_y,win) =
  left = rising_edge (button1);
  right = rising_edge (button2);
  (head,tail,nx,ny,apple_x,apple_y,win) =
    game_loop(max_size,left,right,width,height)
```

Main loop and interaction functions : In the same way as in the previous example, the `-m` option of OCaLustre is used to generate the execution machine code of the program, responsible for repeatedly executing its main node.

Once the option `-m main` is specified, the file `main_io.ml` is generated by OCaLustre with the prototypes of the interaction functions. After being completed, this code essentially contains the functions required for displaying the game, as well as for reading the values of the Arduboy's buttons. However, since the current version of OCaLustre does not handle arrays, the function that checks whether the snake collides with itself (named `eats_itself`) is also defined in this file, directly in OCaml.

8.3.4 Memory consumption

In a 16-bit configuration of the virtual machine, and with the Stop and Copy garbage collector, the Snake game program can run with a stack size of at least 60 values and a heap of at least 744 values, for a memory footprint of 17.1 kilobytes in flash and 2204 bytes of RAM, which comes very close to the 2.5 kilobytes of RAM available on the microcontroller. The use of eager evaluation nevertheless makes it possible to reduce the stack size to 58 values, and the use of the *Mark and Compact* garbage collector allows doubling the allocation zone (and thus reducing the frequency of GC calls) without increasing the size of the RAM. Over the first hundred synchronous instants of the program, the number of garbage collector invocations thus drops from 49 with the *Stop and Copy* GC to 18 with the *Mark and Compact* GC.

Chapter Conclusion

The examples presented in this chapter have demonstrated the feasibility of executing simple, yet complete, programs on devices with very limited resources. These examples also highlight the advantages of using high-level programming paradigms. The programs benefit from the guarantees provided by high-level languages (worst-case execution time estimation, typing, causal loop detection) while still being executable on hardware with very limited resources. Thus, a microcontroller with less than 2.5 kilobytes of RAM is capable of running all the programs described in this chapter.

This very small footprint of OCaLustre programs embedding the OMicroB virtual machine makes it entirely realistic to consider execution on hardware with greater memory resources (such as microcontrollers based on ARM Cortex-M0 processors, which can provide several hundred kilobytes of flash memory and several tens of kilobytes of RAM), thereby enabling the development of richer and more complex programs that can leverage advanced components such as Bluetooth modules, accelerometers, or multicolor touchscreens.

Conclusion and Perspectives

The objective of this thesis was to propose a set of solutions for programming microcontrollers in high-level languages, in order to demonstrate that it is indeed possible to benefit from the advantages of richer programming models while still meeting the constraints of such hardware. To this end, we have described in this manuscript a suite of software solutions and formal approaches that enable a progressive rise in abstraction, providing at each level of this succession of abstractions new guarantees on the programs produced. In this conclusion, we recall the various contributions of this thesis, in light of the increasing levels of abstraction they illustrate and the new guarantees they provide. We then discuss the different perspectives and future work that would be appropriate to further our approach of improving both the safety and expressiveness of microcontroller programming.

Contributions

Figure 9.1 schematically summarizes the positioning of each chapter of this manuscript with respect to the gain in abstraction provided by the different approaches presented in this thesis.

The first contribution of this thesis relied on a *virtual machine approach* enabling the execution of multiparadigm languages on a variety of hardware. We proposed *OMicroB*, a virtual machine for the OCaml language, designed with the goal of being portable to many different models of microcontrollers while maintaining a limited memory footprint. Configurable and optimized, this virtual machine has been run on microcontrollers with memory resources not exceeding 8 kilobytes of RAM and a few tens of kilobytes of flash memory. Moreover, OMicroB's performance in terms of execution speed is quite promising, since it is relatively close to that of the standard OCaml virtual machine, which does not share the same advantages in terms of reduced memory usage. OMicroB also provides a better approach for program debugging : first, because its full compatibility with the bytecode generated by the standard *ocamlc* compiler makes it possible to use traditional OCaml analysis and debugging tools (such as the *ocamldebug* debugger), and second, thanks to its simulator, which can represent the electronic components connected to the programmed microcontroller and test their interactions directly on the developer's computer. Furthermore, the static type safety ensured by the OCaml compiler reduces the number of potential runtime errors. Because of the portability of its bytecode, the expressiveness of the language, and the use of automatic memory management, OMicroB abstracts away the hardware on which the program runs, making it possible to confidently deploy the same program on different devices. Our work on the virtual machine approach and the implementation of OMicroB was presented at the ERTS^2 (*Embedded Real Time Software and Systems*) conference in 2016 [VVC16] and 2018 [VVC18a].

This hardware abstraction provided a first foundation for subsequent abstractions. We then addressed the fact that a very large number of embedded systems, in which microcontrollers play a predominant role, exhibit *concurrent* aspects. This inherent concurrency, resulting from the multiple interactions between

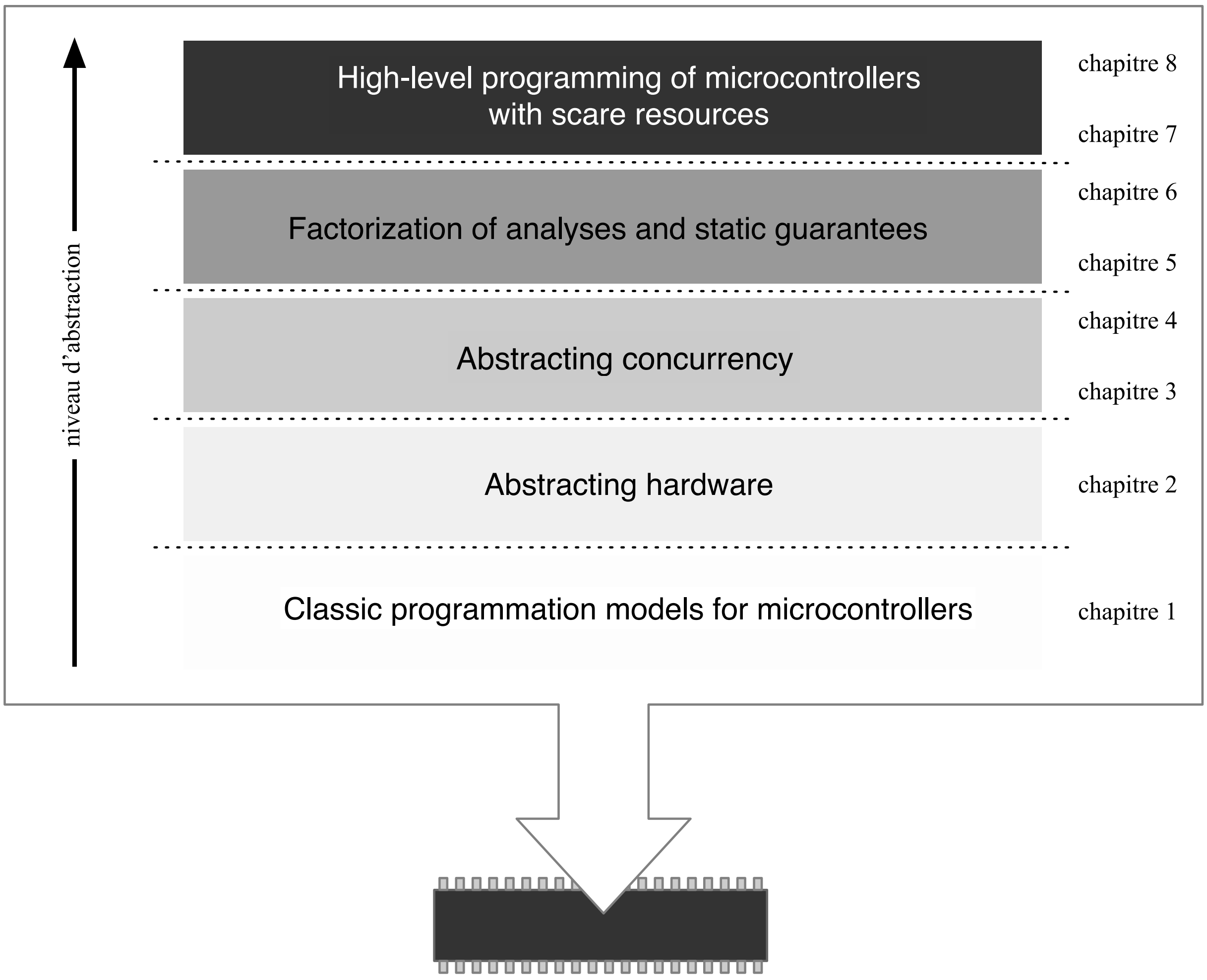

High-level programming of microcontrollers with scare resources
Factorization of analyses and static guarantees
Abstracting concurrency
Abstracting hardware
Classic programmation models for microcontrollers
niveau d'abstraction
chapitre 8
chapitre 7
chapitre 6
chapitre 5
chapitre 4
chapitre 3
chapitre 2
chapitre 1

FIGURE 9.1 – Increasing abstraction for high-level programming of low-resource microcontrollers

such a system and its environment, justifies the addition of a suitable programming model for developing such systems, alongside the more *algorithmic* aspects of general-purpose languages. We therefore proposed *OCaLustre*, an extension of the OCaml language, which makes it possible to use synchronous programming features for the development of embedded programs, while benefiting from the advantages of the host language and the OMicroB virtual machine. This language extension has a compilation model that gives it a small memory footprint, making it well suited to the hardware constraints of the microcontrollers considered. The OCaLustre synchronous extension therefore provides an abstraction of *concurrency* : the use of a synchronous programming model makes the elaboration and interaction of the various concurrent elements of a program implicit. Moreover, the choice of a *dataflow model* makes it straightforward to represent interactions between the program and its physical environment, consisting of electronic components that communicate by emitting electrical signals whose values change over time. This synchronous extension also provides new guarantees for the programs produced, for example by checking at compile time the *causal consistency* of the different software components involved in a program, thereby avoiding deadlocks during execution. In addition, OCaLustre has a formal specification of which several aspects have been mechanically verified using formalization and proof tools. For example, the synchronous clock system, which governs the absence or presence of values during program execution, was formally defined. A tool verifying the consistency between this system and the clock types inferred by the OCaLustre compiler was formalized and proven correct in Coq, then extracted into OCaml code for integration into the compiler. This formal specification of the language and the verification of some of its properties therefore provide OCaLustre programs with a higher level of safety. The description of the OCaLustre language and its compilation was published in the JFLA (*Journées Francophones des Langages Applicatifs*) proceedings in 2017 [VVC17].

The abstractions provided by our work do not come at the expense of verifying essential guarantees for developing embedded, sometimes critical, systems, and even make it possible to *factorize* certain analyses. Indeed, we illustrated the advantage of using a common bytecode across all implementations of the OMicroB virtual machine to perform worst-case execution time (WCET) analysis of a synchronous program. Estimating this execution time is often necessary for the proper behavior of a critical embedded system, to ensure that the synchronous program's reaction time is shorter than the frequency of its inputs. We therefore proposed a method to compute an upper bound of the actual execution time of an OCaLustre program by analyzing its bytecode instructions. This method was then proven correct for a language similar to a subset of OCaml bytecode instructions and implemented in a tool called *Bytecrawler*, which calculates the WCET of an OCaLustre program without requiring users to deal with the complex details of the low-level code actually executed by the virtual machine interpreter. The use of bytecode thus represents, in some sense, a third level of abstraction provided by our solution : this abstraction makes the analyses more straightforward and less dependent on hardware specifics. Our description of WCET analysis, along with its formalization and proof of correctness, was presented at the WCET workshop (*International Workshop on Worst-Case Execution Time Analysis*), a satellite of the ECRTS conference (*Euromicro Conference on Real-Time Systems*), in 2019 [VC19].

The various tools developed during this research, as well as the formal specification of OCaLustre and the partial verification of its associated properties, constitute a complete toolchain for developing safer and richer applications on microcontrollers with very limited hardware resources. Performance measurements evaluating the resource consumption of the OMicroB/OCaLustre pair confirm the viability of our solution, which has low memory usage. In this regard, three complete applications, each based on a different electronic circuit, were described and shown to run on microcontrollers with very scarce resources. The programming of a video game for an *Arduboy* also served as a practical application for a tutorial invited at JFLA 2018 [VVC18b]. All the examples presented further demonstrate the relevance of our various contributions, from the expressive power of the OCaml language to the safety guarantees of the synchronous extension. The ease of deploying such applications is made possible by the portability of the proposed solutions. Indeed, all the software solutions described in this thesis are portable : OCaLustre programs are compiled into standard bytecode, executable on a generic virtual machine, and analyzable with tools that can be easily adapted to different microcontrollers. This portability is at the heart of this thesis's ambitions : to enable the development of safe applications free from the hardware-specific constraints of the target device. The implementation work carried out already makes the different prototypes of the software solutions presented in this thesis fully usable, and industrial applications are envisaged within the framework of the LCHIP project (*Low Cost Integrity Platform*) [⚓11], in partnership with the company CLEARSY. This project aims to build a safe, low-cost execution platform based on PIC32 microcontrollers. Our virtual machine approach and synchronous programming model could provide, within this platform, an alternative execution mode to introduce redundancy in program execution in order to verify correctness.

Perspectives

The various efforts undertaken in this thesis represent a first complete approach to programming microcontrollers using high-level languages and programming models, which provide additional guarantees compared to traditional microcontroller programming techniques. To conclude this manuscript, we propose a set of directions aimed at further increasing the expressive power and guarantees of our solutions, as well as their compatibility with (very) low-resource hardware.

First, the OMicroB virtual machine could benefit from further optimizations to enable, on low-resource hardware, the execution of programs whose memory usage is currently incompatible with our solution. One of the main causes of excessive RAM consumption by an OCaml program comes from immutable values that persist in the heap and are never freed, such as character strings corresponding to exception names (used explicitly or implicitly). For instance, the exceptions `Out_of_memory` and `Stack_overflow` are systematically declared by the runtime library, and their names persist in the heap for the entire execution. Current work aims to move such immutable values from the microcontroller's RAM to its flash memory, which is generally larger but not intended to be modified during program execution. The heap would thus be split between RAM (for mutable values) and flash (for immutable values). This optimization would free up significant heap space for truly dynamic values. Additionally, we plan to continue porting OMicroB to a wider range of hardware, to further validate the portability of our approach. Our targets include ESP32 boards with 520 kilobytes of RAM and 4 megabytes of flash, as well as STM32 microcontrollers on Nucleo boards, whose more constrained models still have 2 kilobytes of RAM and 16 kilobytes of flash.

The initial efforts at formalization and verification of some properties of the OCaLustre language could eventually lead to a fully certified compiler. In particular, proving that compiled programs respect the language semantics would guarantee that no absent flow value is ever read during execution, thereby strengthening the safety of synchronous clock typing. Work is also planned to directly extract from Coq the code implementing most of the compilation steps of an OCaLustre program, in order to ensure that the compilation process adheres to the adopted formal specification. In this respect, we could build upon the work of Bourke *et al.* on Vélus [BBD+17], a Lustre compiler certified in Coq.

Furthermore, new guarantees about the consistency of programs could be checked through the use of *synchronous contracts*, modeled after *model-checking* tools such as Lesar [Rat92] and Kind2 [CMST16]. Initial attempts at defining contracts in OCaLustre with extraction to WhyML (compatible with the Why3 verification platform [FP13]) or to Isabelle/HOL have been undertaken. However, verification quickly becomes difficult when these contracts refer to past flow values (e.g., to express that a flow has increasing values over time), because proving such properties requires a *k-induction* principle that general-purpose verification tools struggle to exploit automatically.

In addition, initial work has aimed to represent OCaLustre's clock typing within OCaml's own type system, using generalized algebraic data types (GADTs). These promising efforts could enable the standard OCaml compiler itself to verify the complete type safety of an OCaLustre programcovering both standard data typing (whose correctness with respect to the OCaml translation we have proven) and clock typing. Leveraging OCaml's rich type system in this way would be a further advantage of using a high-level language.

The specification of OCaLustre and its prototype currently handle only flows of simple types, such as booleans and integers. This restriction was made to preserve the same semantics as Lustre and to support a non-allocating compilation mode compatible with all valid OCaLustre programs. However, it would be advantageous to exploit the richness of the host language by allowing flows of more complex types. Such an extension would make OCaLustre programs more expressive. For example, an OCaLustre program could manipulate flows of tuples, flows of lists, or even flows of functions or nodes, enabling higher-order synchronous programming as in Lucid Synchrone. Extending OCaLustre to richer data types seems straightforward for the standard compilation model discussed in Chapter 4, but it would make WCET verification more difficult because of the dynamic allocation of values it introduces.

Finally, our solution could benefit from more advanced static analyses on OCaml programs, for instance to bound the amount of memory allocated by a program and guarantee that no heap overflow occurs during execution. We envisage drawing inspiration from and extending related work targeting ML-style embedded programming. These approaches introduce an explicit region system into the language in order to statically estimate an upper bound on the maximum number of live values in the heap during execution [SC16a]. Such analyses could also make it possible to integrate GC execution time into WCET computation (potentially by triggering it deliberately, e.g., at the beginning of a synchronous instant).

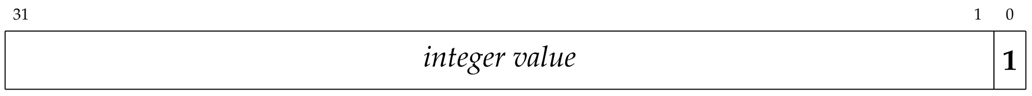
All of these perspectives ultimately aim to continue the approach presented in this thesis : increasing the safety of embedded systems programming while offering higher-level programming models on microcontrollers that still retain very limited resources.

A Representation of OCaml Values

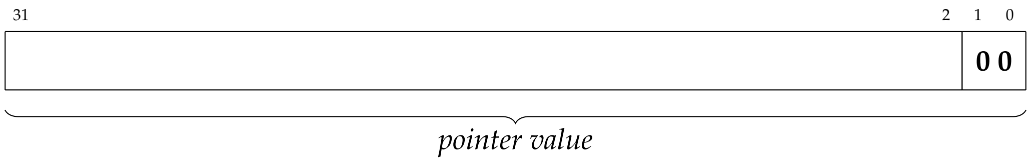
A.1 Representation of Values in the ZAM

A.1.1 32-bit Representation

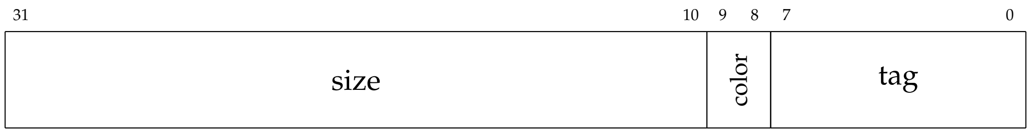
Integer :



Pointer :



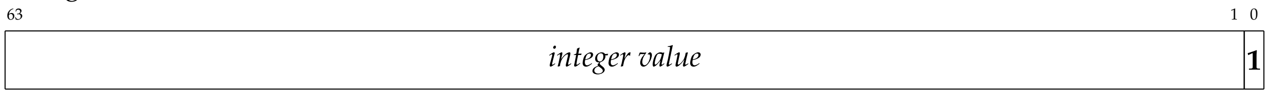
Block header :



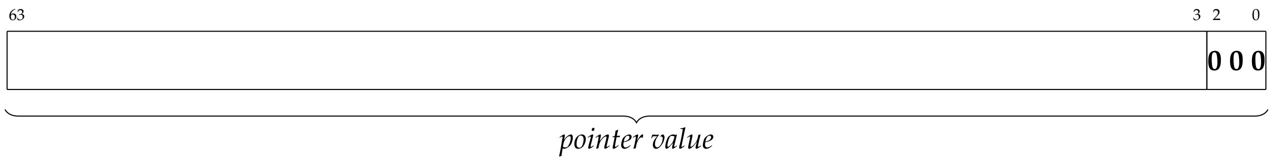
Floating-point numbers are allocated on the heap in blocks that contain two OCaml values (these are double-precision floats on $32 \times 2 = 64$ bits).

A.1.2 64-bit Representation

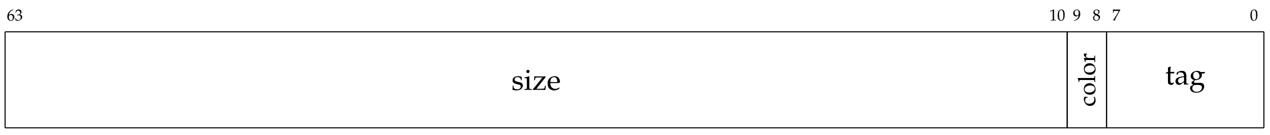
Integer :



Pointer :



Block header :

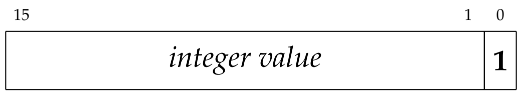


Floating-point numbers are allocated on the heap in blocks that contain one OCaml value (these are also 64-bit double-precision floats).

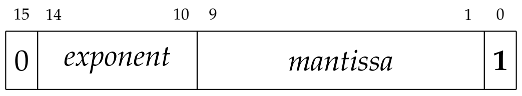
A.2 Representation of Values in OMicroB

A.2.1 16-bit Representation

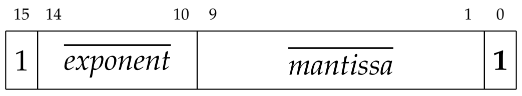
Integer :



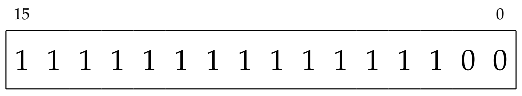
Floating-point (positive) :



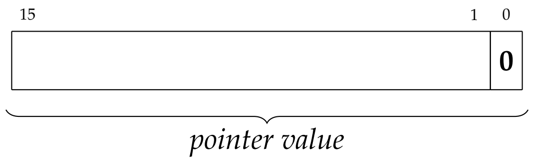
Floating-point (negative) :



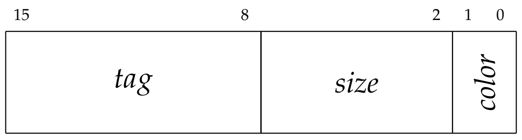
NaN :



Heap pointer :

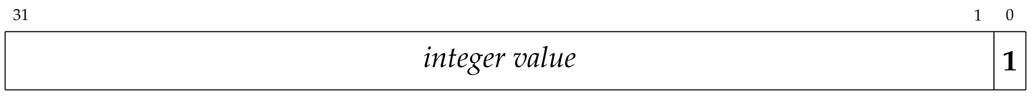


Block header :

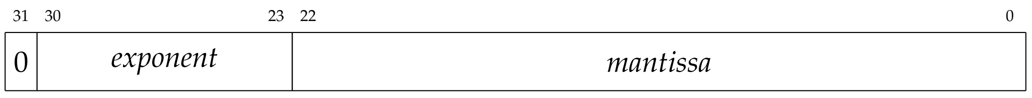


A.2.2 32-bit Representation

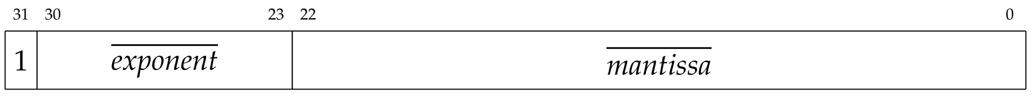
Integer :



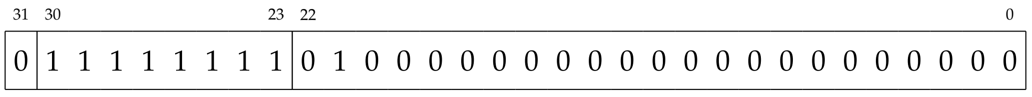
Floating-point (positive) :



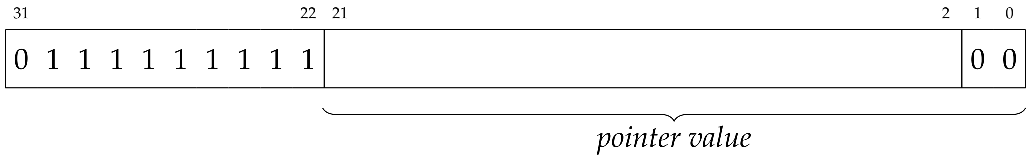
Floating-point (negative) :



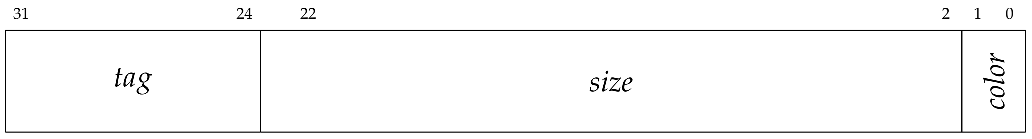
NaN :



Heap pointer :

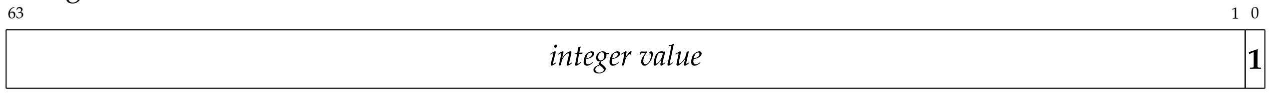


Block header :

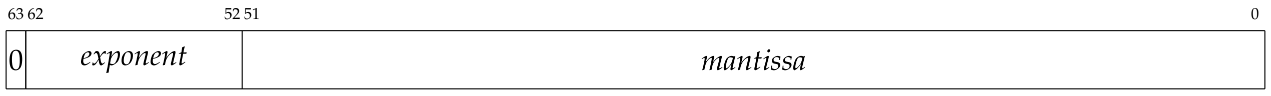


A.2.3 64-bit Representation

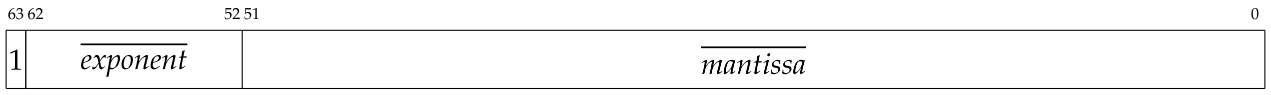
Integer :



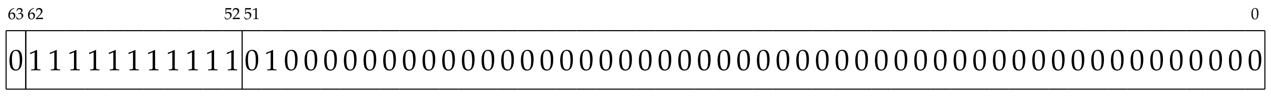
Floating-point (positive) :



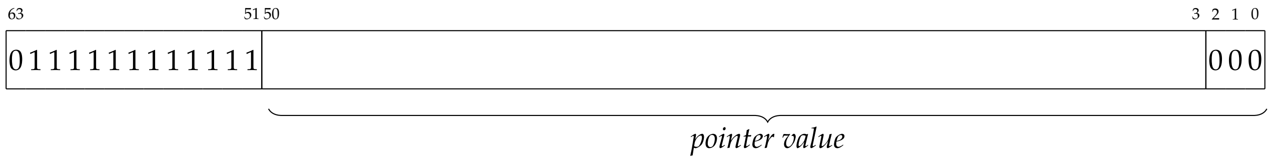
Floating-point (negative) :



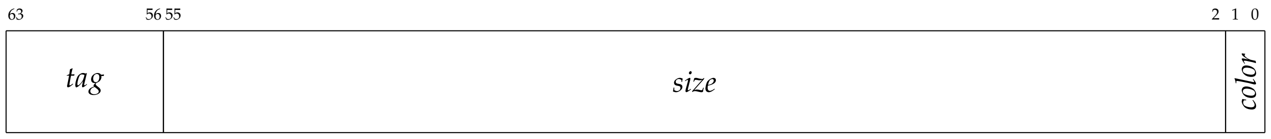
NaN :



Heap pointer :



Block header :



B Lucid Synchrone Code of the Adder

```
let xor a b = if a then not b else b

let node fulladder (a,b,cin) =
  let x = (xor a b) in
  let s = (xor x cin) in
  let and1 = (x && cin) in
  let and2 = (a && b) in
  let cout = (and1 || and2) in
  (s,cout)

let node twobits_adder (c0,a0,a1,b0,b1) =
  let (s0,c1) = fulladder (a0,b0,c0) in
  let (s1,c2) = fulladder (a1,b1,c1) in
  (s0,s1,c2)

let node fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3) =
  let (s0,s1,c2) = twobits_adder (c0,a0,a1,b0,b1) in
  let (s2,s3,c4) = twobits_adder (c2,a2,a3,b2,b3) in
  (s0,s1,s2,s3,c4)

let node heightbits_adder (c0,a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7) =
  let (s0,s1,s2,s3,c4) = fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3) in
  let (s4,s5,s6,s7,c8) = fourbits_adder (c4,a4,a5,a6,a7,b4,b5,b6,b7) in
  (s0,s1,s2,s3,s4,s5,s6,s7,c8)
```

C Performance Measurement Results

Name	**Execution time with ocamlrun** (seconds)	**Execution time with OMicroB** (seconds)	**Ratio**	**Execution speed with OMicroB** (millions of instr./second)	**Number of OMicroB GC activations**
`apply`	1.22	2.14	1.75	306.33	29
`fibo`	0.55	1.30	2.36	376.04	0
`takc`	1.05	3.11	2.96	378.38	110500
`oddeven`	0.28	0.61	2.17	604.60	0
`floats`	0.56	1.05	1.87	219.46	0
`integr`	0.04	0.12	3.00	222.16	21
`eval`	0.04	0.08	2.00	377.50	1153
`sieve`	0.04	0.07	1.75	486.00	1666
`objet`	0.08	0.17	2.12	389.77	3424
`functor`	0.26	0.58	2.23	406.31	10001
`bubble`	0.93	1.51	1.62	473.71	17
`jdlv`	0.36	0.74	2.05	463.82	2999
`share`	0.11	0.27	2.45	444.67	321
`abrsort`	0.47	1.17	2.48	335.00	38084
`queens`	1.37	3.47	2.53	399.24	56666

Table C.1 – Performance measurements with the Mark and Compact GC (on PC)
OMicroB options : -arch 16 -gc MC -stack-size 500 -heap-size 2500

Name	**Execution time with ocamlrun** (seconds)	**Execution time with OMicroB** (seconds)	**Ratio**	**Execution speed with OMicroB** (millions of instr./second)	**Number of OMicroB GC activations**
`apply`	1.21	2.27	1.87	288.79	58
`fibo`	0.54	1.16	2.14	421.43	0
`takc`	1.04	2.94	2.82	400.26	221999
`oddeven`	0.28	0.62	2.21	594.85	0
`floats`	0.59	0.57	0.96	404.28	0
`integr`	0.04	0.06	1.50	440.00	41
`eval`	0.04	0.08	2.00	377.50	2307
`sieve`	0.04	0.07	1.75	486.00	3333
`objet`	0.08	0.16	2.00	414.13	10989
`functor`	0.24	0.60	2.50	392.77	30002
`bubble`	0.93	1.43	1.53	500.21	35
`jdlv`	0.37	0.72	1.94	476.70	6666
`share`	0.11	0.26	2.36	461.77	684
`abrsort`	0.47	1.14	2.42	343.81	115198
`queens`	1.37	3.22	2.35	430.23	149999

Table C.2 – Performance measurements in a 32-bit representation of values (on PC)
OMicroB options : -arch 32 -gc SC -stack-size 500 -heap-size 2500

Name	**Execution time with ocamlrun** (seconds)	**Execution time with OMicroB** (seconds)	**Ratio**	**Execution speed with OMicroB** (millions of instr./second)	**Number of OMicroB GC activations**
`apply`	1.26	2.36	1.87	277.77	58
`fibo`	0.58	1.21	2.08	404.01	0
`takc`	1.06	2.97	2.80	396.21	219666
`oddeven`	0.29	0.60	2.06	614.68	0
`floats`	0.58	0.75	1.29	307.25	0
`integr`	0.05	0.06	1.20	444.33	41
`eval`	0.04	0.08	2.00	377.50	2272
`sieve`	0.05	0.07	1.40	486.00	3333
`objet`	0.08	0.18	2.25	368.12	10666
`functor`	0.24	0.60	2.50	392.77	30002
`bubble`	1.00	1.55	1.55	461.49	34
`jdlv`	0.37	0.83	2.24	413.53	6666
`share`	0.11	0.25	2.27	480.24	675
`abrsort`	0.52	1.20	2.30	326.62	112604
`queens`	1.35	3.29	2.43	421.08	144999

Table C.3 – Performance measurements in a 64-bit representation of values (on PC)
OMicroB options : -arch 64 -gc SC -stack-size 500 -heap-size 2500

Name	**Execution time**	**Speed**	**Number of GC activations**
	(seconds)	(thousands instr./second)	
`apply`	7.860	83.432	0
`fibo`	2.999	163.065	0
`takc`	13.730	85.714	434
`oddeven`	2.262	163.085	0
`floats`	4.780	48.229	0
`integr`	0.379	69.902	0
`eval`	0.355	85.416	4
`sieve`	0.390	87.477	9
`bubble`	7.469	94.519	0
`jdlv`	3.737	91.938	16
`share`	1.373	88.275	1

TABLE C.4 – Program execution speed measurements for OMicroB in 32-bit mode (on ATmega2560)
OMicroB options : -arch 32 -gc SC -stack-size 500 -heap-size 1400

Programs not appearing in this table correspond to those that cannot be executed with this reduced heap size used to compensate for the value size.

D Application Code

D.1 Punch Card Reader Program

```
open Avr

let t = Array.make 8 false
let clk = PIN13
let data = PIN12
let leds = [| PIN42 ; PIN43; PIN44; PIN45; PIN46; PIN47; PIN48; PIN49 |]

let init () =
  (* set input pins *)
  pin_mode clk INPUT;
  pin_mode data INPUT;
  (* set output pins *)
  Array.iter (fun x -> pin_mode x OUTPUT) leds

let input_clk () = bool_of_level (digital_read clk)
let input_data () = bool_of_level (digital_read data)

(* function that switch on/off an LED *)
let update_led i b = digital_write leds.(i) (level_of_bool b)

let output i data clk send =
  if clk then t.(i) ← data;
  if send then Array.iteri update_led t

let%node edge x ~return:e =
  e = (x && (not (true ⋙ x)))

let%node read_bit (top,bot) ~return:(clk,data) =
  clk = edge top;
  data = bot [@when clk]

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

let%node read_card (top,bot) ~return:(i,data,clk,send) =
  (clk,data) = read_bit (top,bot);
  i = count(8 [@when clk]);
  send = merge clk (i = 7) false

let () =
  (* initialising hardware *)
  init ();
  (* create state of the main node *)
  let st = read_card_alloc () in
  while true do
    (* lecture des entrées *)
    let c = input_clk () in
    let d = input_data () in
    (* update state of the main node *)
    read_card_step st c d;
    (* emit outputs *)
    let i = st.read_card_out_i in
    let data = st.read_card_out_data in
    let clk = st.read_card_out_clk in
    let send = st.read_card_out_send in
    output i data clk send
  done
```

D.2 Temperer Program

D.2.1 tempereuse.ml

```
(** switch on/off when + and - are pushed together **)
let%node thermo_on (p,m) ~return:(b) =
  b = (true >>> if p && m then not b else b)

(** change desired temperature dependending on button pushed **)
let%node set_wanted_temp (p,m) ~return:(w) =
  w = (325 >>> if p then w+5 else if m then w-5 else w)

(** main node: calculation of the desired temp and the state of the resistor *)
(** Temps are in 10th of degrees C **)
let%node thermo (plus,minus,real_temp) ~return:(on,wanted,real,resistor) =
  on = thermo_on (plus,minus);
  wanted = set_wanted_temp (plus[@when on], minus[@when on]);
  real = real_temp [@when on];
  heat = (real < wanted);
  resistor = merge on heat false
```

D.2.2 thermo_io.ml

```
open Avr

(* Display *)
let lcd = LiquidCrystal.create4bitmode PIN13 PIN12 PIN18 PIN19 PIN20 PIN21
```

```

(* déclaration des pins *)
let plus = PIN7
let minus = PIN6
let resistor = PIN10
let sensor = PINA0

(* Convert temperature *)
let convert_temp t =
  let f = (float_of_int (1033 - t) /. 11.67) in
  int_of_float (f*.100.)

(* Read temperature *)
let read_temp () =
  let t = analog_read sensor in
  Serial.write_string "an=";
  Serial.write_int t;
  convert_temp t

(* Display temperatures on the LCD *)
let print_temp wanted real =
  let split_temp t =
    let u = t/10 in
    let dec = t mod 10 in
    (u,dec) in
  LiquidCrystal.clear lcd;
  LiquidCrystal.home lcd;
  let (wu,wd) = split_temp wanted in
  let (ru,rd) = split_temp real in
  LiquidCrystal.print lcd "Wanted T :";
  LiquidCrystal.print lcd ((string_of_int wu)^"."^(string_of_int wd));
  LiquidCrystal.setCursor lcd 0 1;
  LiquidCrystal.print lcd "Actual T :";
  LiquidCrystal.print lcd ((string_of_int ru)^"."^(string_of_int rd))

(*** Input/output functions of the synchronous instant ***)
(** set-up function **)
let init_thermo () =
  (* set-up analog read *)
  Avr.adc_init ();
  (* set-up display *)
  LiquidCrystal.lcdBegin lcd 16 2;
  (* set-up pins *)
  pin_mode sensor INPUT;
  pin_mode resistor OUTPUT;
  pin_mode plus INPUT;
  pin_mode minus INPUT
```

```

(** input function **)
let input_thermo () =
  let plus = digital_read plus in
  let minus = digital_read minus in
  let plus = bool_of_level plus in
  let minus = bool_of_level minus in
  let real_temp = read_temp () in
  (plus,minus,real_temp)

(** output function **)
let output_thermo (on,wanted,real,res) =
  if on then
    begin
      print_temp wanted real;
      digital_write resistor (if res then HIGH else LOW)
    end
  else
    begin
      LiquidCrystal.home lcd;
      LiquidCrystal.clear lcd;
      LiquidCrystal.print lcd "..."
    end;
```

D.2.3 Handling the Heating Duty Cycle

The following OCaLustre program accounts for the inertia in the temperature rise of the preparation : it measures the proportion to which the heating element must be activated.

```

let%node min(a,b) ~return:c =
  c = if a < b then a else b

let%node max(a,b) ~return:c =
  c = if a > b then a else b

(** Calculation of the heating proportion (in %) **)
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)))

let%node timer (number) ~return:(alarm) =
  time = (0 ⋙ (time + 10)) mod 100;
  alarm = if (time < number) then true else false

let%node heat (w,c) ~return:(h) =
  count = (0 ⋙ count + 1) mod 10;
  update = (count = 0);
(* The duty cycle (prop) is updated every 10 instants *)
  prop = merge update
           (update_prop (w [@when update],c [@when update]))
           ((0 ⋙ prop) [@whennot update]);
  h = timer (prop)

(** Turn on/off when + and - are pressed simultaneously **)
let%node thermo_on(p,m) ~return:(b) =
  b = (true ⋙ if p && m then (not b) else b)

(** Change the desired temperature depending on the button pressed **)
let%node set_wanted_temp (p,m) ~return:(w) =
  w = (325 ⋙ if p then w+5 else if m then w-5 else w)

(** Main node: computation of the desired temperature and the state of the heating element **)
(** Temperatures are expressed in tenths of degrees Celsius **)
let%node thermo (plus,minus,real_temp) ~return:(on,wanted,real,resistor) =
  on = thermo_on (plus,minus);
  wanted = set_wanted_temp (plus[@when on], minus[@when on]);
  real = real_temp [@when on];
  heat = heat (wanted,real);
  resistor = merge on heat false
```

D.3 Snake Program

D.3.1 spi.ml

```
(*** SPI (Serial Peripheral Interface) connection management module ***)

open Avr

(** Initialize the SPI connection **)
let begin_spi ~sck ~mosi =
  set_bit SPCR MSTR;
  set_bit SPCR SPE;
  set_bit SPSR SPI2x;
  pin_mode sck OUTPUT;
  pin_mode mosi OUTPUT

(** Stop the SPI connection **)
let end_spi () = clear_bit SPCR SPE

(*** Send data via the SPI connection ***)
let transfer data = write_register SPDR data
```

D.3.2 oled.ml

```
open Avr

(*** Write/read functions for the display buffer ***)
external write_buffer : int -> int -> bool -> unit = "caml_buffer_write"
external read_buffer : int -> int -> bool = "caml_buffer_read"
external get_byte_buffer : unit -> int = "caml_buffer_get_byte"

(*** Screen boot program ***)
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 *)
  |]

(*** Send the program to the screen ***)
let transfer_program prog =
  Array.iter Spi.transfer prog

(*** Set the screen to "command" mode ***)
let command_mode cs dc =
  digital_write cs HIGH;
  digital_write dc LOW;
  digital_write cs LOW

(*** Set the screen to "data" mode ***)
let data_mode cs dc =
  digital_write dc HIGH;
  digital_write cs LOW

(*** Send a command to the screen ***)
let send_lcd_command cs dc com =
  command_mode cs dc;
  Spi.transfer com;
  data_mode cs dc

(*** Draw a pixel (true = black / false = white) ***)
let draw x y color =
  write_buffer x y color

(*** Clear the screen ***)
```

```
let clear() =
 for _i = 0 to 1023 do
   Spi.transfer(0x00)
 done

(*** Send the buffer to the screen ***)
let flush () =
  for _i = 0 to 1023 do
    Spi.transfer(get_byte_buffer())
  done

(*** Screen initialization ***)
let boot ~cs ~dc ~rst =
  digital_write rst HIGH;
  digital_write rst LOW;
  digital_write rst HIGH;
  command_mode cs dc;
  transfer_program boot_program;
  data_mode cs dc;
  clear()
```

D.3.3 arduboy.ml

```
open Avr

(** The pins used **)
let cs = PIN12
let dc = PIN4
let rst = PIN6
let button_left = PINA2
let button_right = PINA1
let button_down = PINA3
let button_up = PINA0
let button_a = PIN7
let button_b = PIN8
let blue = PIN9
let red = PIN10
let green = PIN11

(** Initialization of the common-anode RGB LEDs (HIGH = off) **)
let init_led l =
  digital_write l HIGH

(** Turn on one of the common-anode RGB LEDs (LOW = on) **)
let light_led l =
  digital_write l LOW

(** Initialization of the pins **)
```

```
let boot_pins () =
  List.iter (fun x -> pin_mode x INPUT_PULLUP) [button_left; button_right; button_up;
                                                button_down];
  pin_mode button_a INPUT_PULLUP;
  pin_mode button_b INPUT_PULLUP;
  List.iter (fun x -> pin_mode x OUTPUT) [red;green;blue];
  List.iter (fun x -> pin_mode x OUTPUT) [cs;dc;rst];
  List.iter init_led [red;green;blue]

(** Initialization of the pins, the SPI connection, and the screen **)
let init () =
  boot_pins ();
  Spi.begin_spi ~sck:SCK ~mosi:MOSI;
  Oled.boot ~cs:cs ~dc:dc ~rst:rst
```

D.3.4 snake.ml

```

(** Snake direction **)
type direction = South | North | East | West

(** Orientation functions: the snake turns left and right
 ** relative to its current direction **)
let left_of = function
  | South -> East
  | North -> West
  | East -> North
  | West -> South

let right_of = function
  | South -> West
  | North -> East
  | East -> South
  | West -> North

(** Modulo **)
let nmod x y =
  (x + y) mod y

(** Random selection of an integer **)
let new_position n = Random.int n

(** New coordinates **)
let%node new_coord (dir,max,v,dir1,dir2) ~return:n =
  n = if dir = dir1 then call nmod (v-1) max
     else if dir = dir2 then call nmod (v+1) max
     else v

```

```
(** New position of the head of the snake **)
let%node new_head (dir,w,h) ~return:(x,y) =
  x = new_coord (dir,w,0 ⋙ x,West,East);
  y = new_coord (dir,h,0 ⋙ y,North,South)

(** Turn left **)
let%node left (dir) ~return:ndir =
  ndir = call left_of dir

(** Turn right **)
let%node right (dir) ~return:ndir =
  ndir = call right_of dir

(** Direction of the snake based on previous direction **)
let%node direction (l,r) ~return:dir =
  pre_dir = (South ⋙ dir);
  dir = if l then
          (merge l (left (pre_dir [@when l])) (pre_dir [@whennot l]))
        else
          (merge r (right (pre_dir [@when r])) (pre_dir [@whennot r]))

(** New position of the apple **)
let%node new_apple (width,height) ~return:(a_x,a_y) =
  (a_x,a_y) = (call new_position width, call new_position height)

(** Game loop **)
let%node game_loop (max_size,left,right,width,height)
     ~return:(head,tail,new_x,new_y,apple_x,apple_y,win) =
  (new_x,new_y) = new_head (dir,width,height);
  dir = direction(left,right);
  head = (1 ⋙ ((head+1) mod max_size));
  eats = (apple_x = new_x && apple_y = new_y);
  (a_x,a_y) = new_apple (width [@when eats], height [@when eats]);
  apple_x = (10 ⋙ merge eats a_x (apple_x [@whennot eats]));
  apple_y = (10 ⋙ merge eats a_y (apple_y [@whennot eats]));
  tail = merge eats ((0⋙ tail)[@when eats]) (0 ⋙ ((tail+1) mod max_size) [@whennot eats]);
  size = (1 ⋙ merge eats ((size + 1) [@when eats]) (size [@whennot eats]));
  win = (size = max_size -1)

(** Rising edge (for the buttons) **)
let%node rising_edge i ~return:o =
  o = (i && (not (false ⋙ i)))

(** Main node **)
let%node main (max_size,button1,button2,width,height) ~return:(head,tail,nx,ny,apple_x,apple_y,win) =
  left = rising_edge (button1);
  right = rising_edge (button2);
  (head,tail,nx,ny,apple_x,apple_y,win) = game_loop(max_size,left,right,width,height)
```

D.3.5 main_io.ml

```
open Avr

(*** Game functions du jeu ***)

(** The array containing the positions of the snake's body **)
let max_size = 15
let snake = Array.make max_size (0,0)

(** Tests whether the snake collides with itself *)
exception Lose
let eats_itself head tail =
  let f c = if c = snake.(head) then (raise Lose) in
  try
    if tail < head then
      begin
        for i = tail to head - 1 do
          f snake.(i);
        done
      end
    else
      begin
        for i = tail to max_size - 1 do
          f snake.(i);
        done;
        for i = 0 to head - 1 do
          f snake.(i);
        done;
      end;
    false
  with Lose -> true

(*** Display functions ***)

let draw_snake head tail : unit =
  let (x,y) = snake.(head) in
  let (x',y') = snake.(tail) in
  (* draw a new head *)
  Oled.draw x y true;
  (* the tail is erased to give the illusion of movement *)
  Oled.draw x' y' false

let draw_apple x y : unit =
  Oled.draw x y true

```

```
(*** Functions of the synchronous execution engine ***)

(** Initialization of the synchronous program **)
let init_main () = Arduboy.init ()

(** Input function of each synchronous instant **)
let input_main () =
  let l = Avr.digital_read Arduboy.button_left in
  let r = Avr.digital_read Arduboy.button_right in
  (max_size,bool_of_level l,bool_of_level r,64,32)

(** Output function of each synchronous instant **)
let output_main (head,tail,nx,ny,apple_x,apple_y,win) =
  snake.(head) <- (nx,ny);
  draw_snake head tail;
  draw_apple apple_x apple_y;
  Oled.flush ();
  if win then
    begin
      (* Win *)
      Arduboy.light_led Arduboy.green;
      Avr.delay 2000;
      raise Exit
    end
  else if eats_itself head tail then
    begin
      (* Lose *)
      Arduboy.light_led Arduboy.red;
      Avr.delay 2000;
      raise Exit
    end
```

Web References

[⚓1] Annexes électroniques du manuscrit.
https://stevenvar.github.io/these.

[⚓2] Certifications de KCG.
http://www.esterel-technologies.com/products/scade-suite/automatic-code-generation/scade-suite-kcg-certification-kits/.

[⚓3] Description de PPX.
http://ocamllabs.io/doc/ppx.html.

[⚓4] Fiche technique de l'écran SSD1306.
https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf.

[⚓5] Global MCU Market 2018 by Manufacturers, Regions, Type and Application, Forecast to 2023.
https://www.orianresearch.com/request-sample/553889.

[⚓6] Instructions de la machine virtuelle OCaml (site du projet Cadmium).
http://cadmium.x9c.fr/distrib/caml-instructions.pdf.

[⚓7] Page de l'IDE Arduino (site de la plateforme Arduino).
https://www.arduino.cc/en/Main/Software.

[⚓8] Page de l'IDE Atmel Studio (site de Microchip).
https://www.microchip.com/mplab/avr-support/atmel-studio-7.

[⚓9] Page de l'outil ocamlclean (site du projet OCaPIC).
http://www.algo-prog.info/ocapic/web/index.php?id=ocamlclean.

[⚓10] Page du compilateur MPLAB (site de Microchip).
https://www.microchip.com/mplab.

[⚓11] Page du projet LCHIP (site de Clearsy).
http://www.clearsy.com/2016/10/4260/.

[⚓12] Page du projet python-on-a-chip (archive).
https://code.google.com/archive/p/python-on-a-chip/.

[⚓13] Site de la machine virtuelle Java HaikuVM.
http://haiku-vm.sourceforge.net.

[⚓14] Site de la machine virtuelle NanoVM.
http://www.harbaum.org/till/nanovm.

[⚓15] Site de la plateforme Arduino.
https://www.arduino.cc.

[⚓16] Site de la plateforme microEJ.
https://www.microej.com.

[⚓17] Site de la suite logicielle Proteus.
https://www.labcenter.com.

[⚓18] Site de l'implémentation Bigloo.
https://www-sop.inria.fr/mimosa/fp/Bigloo.

[⚓19] Site de Lucid Synchrone.
https://www.di.ens.fr/~pouzet/lucid-synchrone/index.html.

[⚓20] Site de micro :bit.
https://microbit.org/.

[⚓21] Site de ReactiveML.
http://rml.lri.fr.

[⚓22] Site du langage Guile.
https://www.gnu.org/software/guile.

[⚓23] Site du programmateur usbpicprog.
http://usbpicprog.org.

[⚓24] Site du projet AVR-Ada.
https://sourceforge.net/projects/avr-ada.

[⚓25] Site du projet AVR Libc.
https://www.nongnu.org/avr-libc.

[⚓26] Site du projet AVR-Rust.
https://github.com/avr-rust.

[⚓27] Site du projet SDCC.
http://sdcc.sourceforge.net.

[⚓28] Site du projet TinyGO.
https://tinygo.org.

[⚓29] Tutoriel OCaml - Objets (site officiel du langage OCaml) .
https://ocaml.org/learn/tutorials/objects.html.

References

[ABC⁺17] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich et Steve Zdancewic : *Position paper : the science of deep specification*. Philosophical Transactions of the Royal Society A : Mathematical, Physical and Engineering Sciences, 375(2104) :20160331, septembre 2017. https://doi.org/10.1098/rsta.2016.0331. *[cited on page 36]*

[Abr96] Jean-Raymond Abrial : *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996, ISBN 0-521-49619-5. *[cited on page 36]*

[ACR⁺08] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa et Miguel Castro : *Preventing Memory Error Exploits with WIT*. Dans *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pages 263–277, Oakland, CA, USA, mai 2008. IEEE Computer Society. https://doi.org/10.1109/SP.2008.30. *[cited on page 34]*

[AGG07] Elvira Albert, Samir Genaim et Miguel Gómez-Zamalloa : *Heap Space Analysis for Java Bytecode*. Dans *Proceedings of the 6th International Symposium on Memory Management (ISMM 2007)*, pages 105–116, Montréal, Québec, Canada, octobre 2007. ACM. https://doi.org/10.1145/1296907.1296922. *[cited on page 20]*

[AS87] Bowen Alpern et Fred B. Schneider : *Recognizing Safety and Liveness*. Distributed Computing, 2(3) :117–126, septembre 1987, ISSN 1432-0452. https://doi.org/10.1007/BF01782772. *[cited on page 34]*

[Aug13] Cédric Auger : *Compilation certifiée de SCADE/LUSTRE*. Thèse de doctorat, Université Paris-Sud, Orsay, France, 2013. https://tel.archives-ouvertes.fr/tel-00818169. *[2 citations pages 37 et 104]*

[AW77] Edward A. Ashcroft et William W. Wadge : *Lucid, a Nonprocedural Language with Iteration*. Communications of the ACM, 20(7) :519–526, 1977. https://doi.org/10.1145/359636.359715. *[2 citations pages 29 et 75]*

[Bad14] Yusuf Abdullahi Badamasi : *The working principle of an Arduino*. Dans *Proceedings of the 11th International Conference on Electronics, Computer and Computation (ICECCO 2014)*, pages 1–4, Abuja, Nigeria., septembre 2014. https://doi.org/10.1109/ICECCO.2014.6997578. *[cited on page 15]*

[BB91] Albert Benveniste et Gérard Berry : *The synchronous approach to reactive and real-time systems*. Proceedings of the IEEE, 79(9) :1270–1282, septembre 1991, ISSN 0018-9219. https://doi.org/10.1109/5.97297. *[cited on page 27]*

[BBD+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet et Lionel Rieg : *A formally verified compiler for Lustre*. Dans *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, pages 586–601, Barcelone, Espagne, juin 2017. ACM. https://doi.org/10.1145/3062341.3062358. *[3 citations pages 37, 104 et 209]*

[BBP18] Timothy Bourke, Lélio Brun et Marc Pouzet : *Towards a verified Lustre compiler with modular reset*. Dans *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, SCOPES 2018*, pages 14–17, Sankt Goar, Allemagne, mai 2018. ACM. https://doi.org/10.1145/3207719.3207732. *[cited on page 37]*

[BC84] Gérard Berry et Laurent Cosserat : *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*. Dans *Seminar on Concurrency*, tome 197 de *Lecture Notes in Computer Science*, pages 389–448, Carnegie-Mellon University, Pittsburg, PA, USA, juillet 1984. Springer. https://doi.org/10.1007/3-540-15670-4_19. *[cited on page 28]*

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic et Robert de Simone : *The synchronous languages 12 years later*. Proceedings of the IEEE, 91(1) :64–83, 2003. https://doi.org/10.1109/JPROC.2002.805826. *[cited on page 35]*

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt et Gareth Smith : *A trusted mechanised JavaScript specification*. Dans *POPL*, pages 87–100. ACM, 2014. *[cited on page 37]*

[BCHP08] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon et Marc Pouzet : *Clock-directed modular code generation for synchronous data-flow languages*. Dans *Proceedings of the 2008 ACM SIGPLAN/-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*, pages 121–130, Tucson, AZ, USA, juin 2008. ACM. https://doi.org/10.1145/1375657.1375674. *[2 citations pages 105 et 115]*

[BCL08] Niels Brouwers, Peter Corke et Koen Langendoen : *Darjeeling, a Java compatible virtual machine for microcontrollers*. Dans *Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference (Middleware 2008)*, pages 18–23, Leuven, Belgique, décembre 2008. ACM. https://doi.org/10.1145/1462735.1462740. *[cited on page 22]*

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange et Pascal Sainrat : *OTAWA : An Open Toolbox for Adaptive WCET Analysis*. Dans *Proceedings of the 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems (SEUS 2010)*, pages 35–46, Weidhofen/Ybbs, Autriche, 2010. Springer. https://doi.org/10.1007/978-3-642-16256-5_6. *[cited on page 35]*

[BDPR17] Timothy Bourke, Pierre Evariste Dagand, Marc Pouzet et Lionel Rieg : *Vérification de la génération modulaire du code impératif pour Lustre*. Dans *Actes des 28èmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, janvier 2017. https://hal.inria.fr/hal-01403830. *[cited on page 90]*

[Bel17] Charles Bell : *Introducing MicroPython*, pages 27–57. Apress, Berkeley, CA, 2017, ISBN 978-1-4842-3123-4. https://doi.org/10.1007/978-1-4842-3123-4_2. *[cited on page 22]*

[Ber86] Jean-Louis Bergerand : *LUSTRE : un langage déclaratif pour le temps réel*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, 1986. *[cited on page 73]*

[BFL18] Clément Ballabriga, Julien Forget et Giuseppe Lipari : *Symbolic WCET computation*. ACM Transactions on Embedded Computing Systems (TECS), 17(2) :39, 2018. *[cited on page 163]*

[BN94] Swagato Basumallick et Kelvin Nilsen : *Cache issues in real-time systems*. Dans *In Proceedings of the 1994 ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time System*, Orlando, FL, USA, juin 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.3755. *[cited on page 155]*

[Bou98] Hédi Boufaied : *Machines d'exécution pour langages synchrones*. Thèse de doctorat, Université de Nice-Sophia Antipolis, Nice, France, 1998. *[cited on page 121]*

[BP13] Timothy Bourke et Marc Pouzet : *Zélus : a synchronous language with ODEs*. Dans *Proceedings of the 16th international conference on Hybrid systems : computation and control (HSCC 2013)*, pages 113–118, Philadelphia, PA, USA, avril 2013. ACM. https://doi.org/10.1145/2461328.2461348. *[cited on page 33]*

[BP19] Timothy Bourke et Marc Pouzet : *Clocked arguments in a verified Lustre compiler*. Dans *Actes des 30èmes Journées Francophones des Langages Applicatifs (JFLA 2019)*, page 16, Les Rousses, France, janvier 2019. https://hal.inria.fr/hal-02005639. *[cited on page 133]*

[BRS18] Frédéric Bour, Thomas Refis et Gabriel Scherer : *Merlin : a language server for OCaml (experience report)*. Proceedings of the ACM on Programming Languages, 2(ICFP) :103 :1–103 :15, septembre 2018. https://doi.org/10.1145/3236798. *[cited on page 98]*

[BSR12] Thomas W. Barr, Rebecca Smith et Scott Rixner : *Design and Implementation of an Embedded Python Run-Time System*. Dans *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 297–308, Boston, MA, USA, juin 2012. USENIX Association. https://www.usenix.org/conference/atc12. *[cited on page 22]*

[BV97] Ross Bannatyne et Greg Viot : *Introduction to microcontrollers*. Dans *Proceedings of the 1997 Western Electronics Show and Convention (WESCON/97)*, pages 564–574, Santa Clara, CA, USA, novembre 1997. https://doi.org/10.1109/WESCON.1997.632384. *[cited on page 13]*

[Cam16] Giulio Camilleri : *An LLVM Bitcode Interpreter for 16-bit MSP430 Microcontrollers*. Mémoire de licence, Faculty of ICT - University of Malta, mai 2016. https://www.um.edu.mt/library/oar/handle/123456789/13777. *[cited on page 21]*

[CH86] Paul Caspi et Nicolas Halbwachs : *A Functional Model for Describing and Reasoning About Time Behaviour of Computing Systems*. Acta Informatica, 22(6) :595–627, 1986. https://doi.org/10.1007/BF00263648. *[cited on page 30]*

[Cha92] Emmanuel Chailloux : *An Efficient Way of Compiling ML to C*. Dans *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 37–51, San Francisco, CA, USA, juin 1992. ACM. *[cited on page 69]*

[CHP06] Paul Caspi, Grégoire Hamon et Mark Pouzet : *Lucid Synchrone, un langage de programmation des systèmes réactifs*. Systèmes Temps-réel : Techniques de Description et de Vérification Théorie et Outils, 1 :217260, 2006. *[cited on page 33]*

[CL14] Raphaëlle Crubillé et Ugo Dal Lago : *On Probabilistic Applicative Bisimulation and Call-by-Value λ-Calculi*. Dans *Proceedings of the 23rd European Symposium on Programming (ESOP 2014)*, tome 8410 de *Lecture Notes in Computer Science*, pages 209–228, Grenoble, France, avril 2014. Springer. https://doi.org/10.1007/978-3-642-54833-8_12. *[cited on page 148]*

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel et Cesare Tinelli : *The Kind 2 Model Checker*. Dans *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016)*, pages 510–517, Toronto, Canada, juillet 2016. Springer International Publishing. https://doi.org/10.1007/978-3-319-41540-6_29. *[cited on page 209]*

[CP99] Paul Caspi et Marc Pouzet : *Lucid Synchrone : une extension fonctionnelle de Lustre*. Dans *Actes des 10èmes Journées Francophones des Langages Applicatifs (JFLA 99)*, Avoriaz, France, février 1999. INRIA. https://hal.archives-ouvertes.fr/hal-01574464. *[2 citations pages 33 et 75]*

[CP01] Antoine Colin et Isabelle Puaut : *A Modular & Retargetable Framework for Tree-Based WCET Analysis*. Dans *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*, pages 37–44, Delft, Pays-Bas, juin 2001. IEEE Computer Society. https://doi.org/10.1109/EMRTS.2001.933995. *[cited on page 35]*

[CP03] Jean-Louis Colaço et Marc Pouzet : *Clocks as First Class Abstract Types*. Dans *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, tome 2855 de *Lecture Notes in Computer Science*, pages 134–155, Philadelphia, PA, USA, octobre 2003. Springer. https://doi.org/10.1007/978-3-540-45212-6_10. *[4 citations pages 86, 98, 114 et 133]*

[CP04] Jean-Louis Colaço et Marc Pouzet : *Type-based initialization analysis of a synchronous dataflow language*. International Journal on Software Tools for Technology Transfer (STTT), 6(3) :245–255, 2004. https://doi.org/10.1007/s10009-004-0160-y. *[cited on page 77]*

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs et John Plaice : *Lustre : A Declarative Language for Programming Synchronous Systems*. Dans *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL 87)*, pages 178–188, Munich, Germany, janvier 1987. ACM Press. https://doi.org/10.1145/41625.41641. *[3 citations pages 29, 73 et 101]*

[CPP17] Jean-Louis Colaço, Bruno Pagano et Marc Pouzet : *SCADE 6 : A formal language for embedded critical software development (invited paper)*. Dans *Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, pages 1–11, Sophia Antipolis, France, septembre 2017. IEEE Computer Society. https://doi.org/10.1109/TASE.2017.8285623. *[3 citations pages 10, 32 et 115]*

[Cri92] Régis Cridlig : *An Optimizing ML to C Compiler*. Dans *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 28–36, San Francisco, CA, USA, juin 1992. ACM. *[cited on page 69]*

[DF05] Danny Dubé et Marc Feeley : *BIT : A Very Compact Scheme System for Microcontrollers*. Higher-Order and Symbolic Computation, 18(3-4) :271–298, 2005. https://doi.org/10.1007/s10990-005-4877-4. *[cited on page 23]*

[DM82] Luis Damas et Robin Milner : *Principal Type-schemes for Functional Programs*. Dans *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212, Albuquerque, New Mexico, janvier 1982. ACM, ISBN 0-89791-065-6. http://doi.acm.org/10.1145/582153.582176. *[cited on page 86]*

[Dor08] Francois Xavier Dormoy : *Scade 6 : a model based solution for safety critical software development*. Dans *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS 2008)*, pages 1–9, Toulouse, France, janvier 2008. *[cited on page 83]*

[DRM11] Gwenaël Delaval, Eric Rutten et Hervé Marchand : *Intégration de la synthèse de contrôleurs discrets dans un langage de programmation*. Dans *Actes du 8ème colloque francophone sur la modélisation des systèmes réactifs (MSR'11)*, Lille, France, novembre 2011. https://hal.inria.fr/inria-00629104. *[cited on page 33]*

[DS00] Stephan Diehl et Peter Sestoft : *Abstract Machines for Programming Language Implementation*. Future Generation Computer Systems, 16(7) :739–751, mai 2000, ISSN 0167-739X. http://doi.org/10.1016/S0167-739X(99)00088-6. *[cited on page 19]*

[FD03] Marc Feeley et Danny Dubé : *PICBIT : A Scheme system for the PIC microcontroller*. Dans *Proceedings of the 4th Workshop on Scheme and Functional Programming*, pages 7–15, Boston, MA, USA, novembre 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.5903. *[cited on page 23]*

[FP13] Jean-Christophe Filliâtre et Andrei Paskevich : *Why3 - Where Programs Meet Provers*. Dans *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, tome 7792, pages 125–128, Rome, Italie, mars 2013. Springer. https://doi.org/10.1007/978-3-642-37036-6_8. *[cited on page 209]*

[Gér13] Léonard Gérard : *Programmer le parallélisme avec des futures en Heptagon un langage synchrone flot de données et étude des réseaux de Kahn en vue dune compilation synchrone*. Thèse de doctorat, Université Paris Sud - Paris XI, septembre 2013. https://tel.archives-ouvertes.fr/tel-00929932. *[2 citations pages 75 et 86]*

[GG87] Thierry Gautier et Paul Le Guernic : *SIGNAL : A declarative language for synchronous programming of real-time systems*. Dans *Proceedings of Functional Programming Languages and Computer Architecture*, tome 274 de *Lecture Notes in Computer Science*, pages 257–277, Portland, Oregon, USA, septembre 1987. Springer. https://doi.org/10.1007/3-540-18317-5_15. *[cited on page 31]*

[GGPP12] Léonard Gérard, Adrien Guatto, Cédric Pasteur et Marc Pouzet : *A modular memory optimization for synchronous data-flow languages : application to arrays in a lustre compiler*. Dans *Proceedings of the 2012 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '12)*, pages 51–60, Pékin, Chine, juin 2012. ACM. https://doi.org/10.1145/2248418.2248426. *[cited on page 33]*

[GHKT14] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai et Xavier Thirioux : *Testing-Based Compiler Validation for Synchronous Languages*. Dans *Proceedings of the 6th International Symposium on NASA Formal Methods (NFM 2014), Houston*, tome 8430 de *Lecture Notes in Computer Science*, pages 246–251, Houston, TX, USA, avril 2014. Springer. https://doi.org/10.1007/978-3-319-06200-6_19. *[cited on page 115]*

[GPPT16] Fei Guan, Long Peng, Luc Perneel et Martin Timmerman : *Open source FreeRTOS as a case study in real-time operating system evolution*. Journal of Systems and Software, 118 :19–35, 2016. https://doi.org/10.1016/j.jss.2016.04.063. *[cited on page 26]*

[Gud93] David Gudeman : *Representing Type Information in Dynamically Typed Languages*. Rapport technique 93-27, University of Arizona, Phoenix, AZ, USA, octobre 1993. *[cited on page 63]*

[Gut97] Scott B. Guthery : *Java Card (Industry Report)*. IEEE Internet Computing, 1(1) :57–59, 1997. https://doi.org/10.1109/4236.585173. *[cited on page 22]*

[Hal03] Dean W. Hall : *PyMite : A Flyweight Python Interpreter for 8-bit Architectures*. Dans *Proceedings of the First Python Community Conference (PyCon 2003)*, pages 26–28, Washington, DC, USA, mars 2003. http://ftp.ntua.gr/mirror/python/pycon/papers/pymite/index.html. *[cited on page 22]*

[Hal05] Nicolas Halbwachs : *A synchronous language at work : the story of Lustre*. Dans *Proceedings of the 3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005)*, pages 3–11, Verona, Italie, juillet 2005. IEEE Computer Society. https://doi.org/10.1109/MEMCOD.2005.1487884. *[cited on page 29]*

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond et Daniel Pilaud : *The synchronous data flow programming language LUSTRE*. Proceedings of the IEEE, 79(9) :1305–1320, Sep. 1991. https://doi.org/10.1109/5.97300. *[cited on page 73]*

[HG16] Caleb Helbling et Samuel Z. Guyer : *Juniper : a functional reactive programming language for the Arduino*. Dans *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016)*, pages 8–16, Nara, Japon, septembre 2016. ACM. https://doi.org/10.1145/2975980.2975982. *[cited on page 9]*

[HK07] Trevor Harmon et Raymond Klefstad : *A Survey of Worst-Case Execution Time Analysis for Real-Time Java*. Dans *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–8, Long Beach, CA, USA, mars 2007. IEEE Computer Society. *[cited on page 163]*

[HPK$^+$09] Kirak Hong, Jiin Park, Taekhoon Kim, Sungho Kim, Hwangho Kim, Yousun Ko, Jongtae Park, Bernd Burgstaller et Bernhard Scholz : *TinyVM, an efficient virtual machine infrastructure for sensor networks*. Dans *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems (SenSys 2009)*, pages 399–400, Berkeley, CA, USA, novembre 2009. ACM. https://doi.org/10.1145/1644038.1644121. *[cited on page 22]*

[HR01] Nicolas Halbwachs et Pascal Raymond : *A tutorial of Lustre*. 2001. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.872. *[cited on page 105]*

[HS02] Niklas Holsti et Sam Saarinen : *Status of the Bound-T WCET tool*. Dans *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, Vienne, Autriche, June 2002. http://www.cs.york.ac.uk/rts/wcet2002. *[cited on page 155]*

[JGS93] Neil D. Jones, Carsten K. Gomard et Peter Sestoft : *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993, ISBN 0-13-020249-5. *[cited on page 68]*

[Jon97] Mike Jones : *What really happened on Mars Rover Pathfinder*. ACM Forum on Risks to the Public in Computers and Related Systems, 19(49), 1997. http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html. *[cited on page 26]*

[JRH19] Erwan Jahier, Pascal Raymond et Nicolas Halbwachs : *The Lustre V6 Reference Manual*. 2019. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf. *[cited on page 75]*

[JWC08] Jeong Hoon Ji, Gyun Woo et Hwan Gue Cho : *A Plagiarism Detection Technique for Java Program Using Bytecode Analysis*. Dans *Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology*, tome 1, pages 1092–1098. IEEE Computer Society, novembre 2008. https://doi.org/10.1109/ICCIT.2008.267. *[cited on page 20]*

[Kah74] Gilles Kahn : *The Semantics of Simple Language for Parallel Programming*. Dans *Proceedings of IFIP Congress 1974*, pages 471–475, Stockholm, Suède, août 1974. *[cited on page 30]*

[Kat10] Yoshiharu Kato : *Splish : A Visual Programming Environment for Arduino to Accelerate Physical Computing Experiences*. Dans *Proceedings of the 8th International Conference on Creating, Connecting and Collaborating through Computing (C^5 2010)*, pages 3–10, La Jolla, CA, USA, janvier 2010. IEEE Computer Society. https://doi.org/10.1109/C5.2010.20. *[cited on page 9]*

[KEH⁺09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch et Simon Winwood : *seL4 : formal verification of an OS kernel*. Dans *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009 (SOSP '09)*, pages 207–220, Big Sky, MT, USA, 2009. ACM. http://doi.acm.org/10.1145/1629575.1629596. *[cited on page 36]*

[KLW14] Robbert Krebbers, Xavier Leroy et Freek Wiedijk : *Formal C Semantics : CompCert and the C Standard*. Dans *Proceedings of the 5th International Conference on Interactive Theorem Proving (ITP 2014)*, tome 8558 de *Lecture Notes in Computer Science*, pages 543–548, Vienne, Autriche, juillet 2014. Springer. https://doi.org/10.1007/978-3-319-08970-6_36. *[cited on page 36]*

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish et Scott Owens : *CakeML : A Verified Implementation of ML*. Dans *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, pages 179–191, San Diego, CA, USA, 2014. ACM, ISBN 978-1-4503-2544-8. http://doi.acm.org/10.1145/2535838.2535841. *[cited on page 36]*

[Kri07] Jean-Louis Krivine : *A call-by-name lambda-calculus machine*. Higher-Order and Symbolic Computation, 20(3) :199–207, 2007. https://doi.org/10.1007/s10990-007-9018-9. *[cited on page 47]*

[Lam77] Leslie Lamport : *Proving the Correctness of Multiprocess Programs*. IEEE Trans. Software Eng., 3(2) :125–143, 1977. https://doi.org/10.1109/TSE.1977.229904. *[cited on page 34]*

[Ler90] Xavier Leroy : *The ZINC experiment : an economical implementation of the ML language*. Rapport technique 117, INRIA, Rocquencourt, France, février 1990. https://hal.inria.fr/inria-00070049/file/RT-0117.pdf. *[2 citations pages 23 et 47]*

[LF08] Francesco Logozzo et Manuel Fähndrich : *On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis*. Dans *Proceedings of the 17th International Conference on Compiler Construction (CC 2008)*, tome 4959 de *Lecture Notes in Computer Science*, pages 197–212, Budapest, Hongrie, mars 2008. Springer. https://doi.org/10.1007/978-3-540-78791-4_14. *[cited on page 20]*

[LHS10] Michael W. Lew, Thomas B. Horton et Mark Sherriff : *Using LEGO MINDSTORMS NXT and LEJOS in an Advanced Software Engineering Course*. Dans *Proceedings of the 23rd IEEE Conference on Software Engineering Education and Training (CSEE&T 2010)*, pages 121–128, Pittsburgh, Pennsylvania, USA, mars 2010. https://doi.org/10.1109/CSEET.2010.31. *[cited on page 22]*

[LL05] Benjamin Livshits et Monica S. Lam : *Finding Security Vulnerabilities in Java Applications with Static Analysis*. Dans *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, USA, 2005. USENIX Association. https://www.usenix.org/conference/14th-usenix-security-symposium/. *[cited on page 20]*

[LM97] Yau-Tsun Steven Li et Sharad Malik : *Performance analysis of embedded software using implicit path enumeration*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 16(12) :1477–1487, 1997. https://doi.org/10.1109/43.664229. *[cited on page 35]*

[Lop09] Bruno Cardoso Lopes : *Understanding and writing an LLVM compiler back-end (tutorial)*. Dans *ELC'09 : Embedded Linux Conference*, San Francisco, CA, USA, avril 2009. *[cited on page 21]*

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha et Alex Buckley : *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st édition, 2014, ISBN 013390590X, 9780133905908. *[cited on page 21]*

[MDLM18] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne et Gilles Muller : *Usuba : Optimizing & Trustworthy Bitslicing Compiler*. Dans *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing (WPMVP@PPoPP 2018)*, pages 4 :1–4 :8, Vienne, Autriche, février 2018. ACM. https://doi.org/10.1145/3178433.3178437. *[cited on page 115]*

[Mil78] Robin Milner : *A theory of type polymorphism in programming*. Journal of Computer and System Sciences, 17(3) :348 – 375, 1978, ISSN 0022-0000. https://doi.org/10.1016/0022-0000(78)90014-4. *[2 citations pages 114 et 133]*

[MP05] Louis Mandel et Marc Pouzet : *ReactiveML : a reactive extension to ML*. Dans *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 82–93, Lisbonne, Portugal, juillet 2005. ACM. https://doi.org/10.1145/1069774.1069782. *[cited on page 28]*

[MPP10] Louis Mandel, Florence Plateau et Marc Pouzet : *Lucy-n : a n-Synchronous Extension of Lustre*. Dans *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, tome 6120 de *Lecture Notes in Computer Science*, pages 288–309, Québec city, Québec, Canada, juin 2010. Springer. https://doi.org/10.1007/978-3-642-13321-3_17. *[cited on page 30]*

[MS17] Mahesh M et Sivraj P : *DrawCode : Visual tool for programming microcontrollers*. Dans *Proceedings of the 3rd International Conference on Advances in Computing,Communication Automation (ICACCA 2017)*, pages 1–6, Dehradun, Inde, septembre 2017. https://doi.org/10.1109/ICACCAF.2017.8344708. *[cited on page 9]*

[MV13] Michel Mauny et Benoît Vaugon : *OCamlCC - Traduire OCaml en C en passant par le bytecode*. Dans *Actes des 24èmes Journées Francophones des Langages Applicatifs (JFLA 2013)*, Aussois, France, février 2013. https://hal.inria.fr/hal-00779721. *[cited on page 69]*

[MV14] Michel Mauny et Benoît Vaugon : *Nullable Type Inference*. Dans *Proceedings of the 2014 OCaml Users and Developers Workshop (OCaml 2014)*, Gothenbourg, Suède, septembre 2014. https://hal.inria.fr/hal-01413294. *[cited on page 118]*

[NPW02] Tobias Nipkow, Lawrence C. Paulson et Markus Wenzel : *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, tome 2283 de *Lecture Notes in Computer Science*. Springer, 2002, ISBN 978-3-540-45949-1. https://doi.org/10.1007/3-540-45949-9. *[cited on page 36]*

[PAM$^+$09] Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury et Jean-Louis Colaço : *Experience report : Using Objective Caml to develop safety-critical embedded tools in a certification framework*. Dans *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 215–220, Édimbourg, Royaume-Uni, août 2009. ACM. https://doi.org/10.1145/1596550.1596582. *[cited on page 32]*

[PB19] Basile Pesin et Julien Bissey : *Programmation de haut niveau pour applications sur microcontrôleurs*. Rapport de travaux encadrés de recherche, Sorbonne Université, juin 2019. *[cited on page 70]*

[Pes18] Basile Pesin : *Paradigmes de programmation alternatifs sur microcontrôleurs via l'OMicroB*. Rapport de stage, Sorbonne Université, juin 2018. *[cited on page 64]*

[PFB$^+$11] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla et David Lesens : *Multi-task Implementation of Multi-periodic Synchronous Programs*. Discrete Event Dynamic Systems, 21(3) :307–338, Sep 2011, ISSN 1573-7594. https://doi.org/10.1007/s10626-011-0107-x. *[cited on page 27]*

[Pie02] Benjamin C. Pierce : *Types and programming languages*. MIT Press, 2002, ISBN 978-0-262-16209-8. *[cited on page 34]*

[Pla88] John Plaice : *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, 1988. *[cited on page 105]*

[Pla89] John Plaice : *Nested clocks : The LUSTRE synchronous dataflow language*. Dans *Proceedings of the 1989 International Symposium on Lucid and Intensional Programming*, pages 1–17, 1989. *[cited on page 83]*

[Pou06] Marc Pouzet : *Lucid Synchrone - Tutorial and Reference Manual*. 2006. `https://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf`. *[cited on page 33]*

[PSS98] Amir Pnueli, Michael Siegel et Eli Singerman : *Translation Validation*. Dans *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*, pages 151–166, Lisbonne, Portugal, mars 1998. Springer. `https://doi.org/10.1007/BFb0054170`. *[cited on page 133]*

[Rat92] Christophe Ratel : *Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE*. Thèse de doctorat, Université Joseph Fourier, Grenoble, France, 1992. `https://tel.archives-ouvertes.fr/tel-00341223`. *[cited on page 209]*

[Reg12] Pablo Vieira Rego : *Integrating 8-bit AVR Micro-Controllers in Ada*. The Ada User Journal, 33(4) :301, 2012. *[cited on page 21]*

[Rey98] John C. Reynolds : *Definitional Interpreters for Higher-Order Programming Languages*. Higher-Order and Symbolic Computation, 11(4) :363–397, 1998. *[cited on page 40]*

[RP19] Mario Aldea Rivas et Héctor Pérez : *Leveraging real-time and multitasking Ada capabilities to small microcontrollers*. Journal of Systems Architecture - Embedded Systems Design, 94 :32–41, 2019. `https://doi.org/10.1016/j.sysarc.2019.02.015`. *[cited on page 21]*

[RWT$^+$06] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger et Bernd Becker : *A Definition and Classification of Timing Anomalies*. Dans *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET '06)*, tome 4 de *OpenAccess Series in Informatics (OASIcs)*, Dresden, Allemagne, juillet 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `https://doi.org/10.4230/OASIcs.WCET.2006.671`. *[cited on page 155]*

[SC16a] Jérémie Salvucci et Emmanuel Chailloux : *Memory Consumption Analysis for a Functional and Imperative Language*. Dans *Proceedings of the 1st international workshop on Resource Aware Computing (RAC 2016)*, tome 330 de *Electronic Notes in Theoretical Computer Science*, pages 27 – 46, Eindhoven, Pays-Bas, avril 2016. `https://hal.sorbonne-universite.fr/hal-01420298`. *[cited on page 209]*

[SC16b] Rémy El Sibaïe et Emmanuel Chailloux : *Synchronous-reactive web programming*. Dans *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2016)*, pages 9–16, Amsterdam, Pays-Bas, novembre 2016. ACM. `https://doi.org/10.1145/3001929.3001931`. *[cited on page 29]*

[Sen07] Koushik Sen : *Concolic testing*. Dans *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 571–572, Atlanta, Georgia, USA, novembre 2007. ACM. `https://doi.org/10.1145/1321631.1321746`. *[cited on page 163]*

[SF09] Vincent St-Amour et Marc Feeley : *PICOBIT : A Compact Scheme System for Microcontrollers*. Dans *Revised Selected Papers of the 21st Internation Symposium on Implementation and Application of Functional Languages (IFL 2009)*, tome 6041 de *Lecture Notes in Computer Science*, pages 1–17, South Orange, NJ, USA, septembre 2009. Springer. https://doi.org/10.1007/978-3-642-16478-1_1. *[cited on page 23]*

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent Nana et Lionel Marcé : *Cheddar : a flexible real time scheduling framework*. Dans *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada*, pages 1–8, Atlanta, GA, USA, novembre 2004. ACM. https://doi.org/10.1145/1032297.1032298. *[cited on page 25]*

[SN08] Konrad Slind et Michael Norrish : *A Brief Overview of HOL4*. Dans *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, tome 5170 de *Lecture Notes in Computer Science*, pages 28–32, Montréal, Québec, Canada, août 2008. Springer. https://doi.org/10.1007/978-3-540-71067-7_6. *[cited on page 36]*

[SNO$^+$10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar et Rok Strnisa : *Ott : Effective tool support for the working semanticist*. Journal of Functional Programming, 20(1) :71–122, 2010. https://doi.org/10.1017/S0956796809990293. *[cited on page 90]*

[SP06] Martin Schoeberl et Rasmus Pedersen : *WCET Analysis for a Java Processor*. Dans *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '06)*, pages 202–211, Paris, France, 2006. ACM, ISBN 1-59593-544-4. http://doi.acm.org/10.1145/1167999.1168033. *[cited on page 20]*

[Spa17] Philip Sparks : *The route to a trillion devices*. rapport technique, ARM, juin 2017. *[cited on page 8]*

[Spi89] Michael Spivey : *An introduction to Z and formal specifications*. Software Engineering Journal, 4(1) :40–50, 1989. *[cited on page 36]*

[SSD$^+$17] Anatoliy Shamraev, Elena Shamraeva, Anatoly Dovbnya, Andriy Kovalenko et Oleg Ilyunin : *Green Microcontrollers in Control Systems for Magnetic Elements of Linear Electron Accelerators*, pages 283–305. Springer International Publishing, Cham, 2017, ISBN 978-3-319-44162-7. https://doi.org/10.1007/978-3-319-44162-7_15. *[cited on page 10]*

[STD85] *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985, pages 1–20, octobre 1985. http://doi.org/10.1109/IEEESTD.1985.82928. *[cited on page 62]*

[Tea19] The Coq Development Team : *The Coq Proof Assistant, version 8.9.0*, janvier 2019. https://doi.org/10.5281/zenodo.2554024. *[3 citations pages 10, 36 et 90]*

[VB14] Jérôme Vouillon et Vincent Balat : *From Bytecode to JavaScript : the Js_of_ocaml Compiler*. Software : Practice and Experience, 44(8) :951–972, 2014. https://doi.org/10.1002/spe.2187. *[cited on page 29]*

[VC19] Steven Varoumas et Tristan Crolard : *WCET of OCaml Bytecode on Microcontrollers : An Automated Method and Its Formalisation*. Dans *Proceedings of the 19th International Workshop on*

Worst-Case Execution Time Analysis (WCET 2019), tome 72 de *OpenAccess Series in Informatics (OASICs)*, pages 5 :1–5 :12. Schloss Dagstuhl, juillet 2019. https://doi.org/10.4230/OASIcs.WCET.2019.5. *[cited on page 207]*

[VVC16] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *Concurrent Programming of Microcontrollers, a Virtual Machine Approach*. Dans *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS^2 2016)*, Toulouse, France, 2016. https://hal.archives-ouvertes.fr/ERTS2016. *[2 citations pages 26 et 205]*

[VVC17] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs*. Dans *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, 2017. https://hal.archives-ouvertes.fr/JFLA2017. *[cited on page 207]*

[VVC18a] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *A Generic Virtual Machine Approach for Programming Microcontrollers : the OMicroB Project*. Dans *Proceedings of the 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, 2018. https://hal.archives-ouvertes.fr/ERTS2018. *[cited on page 205]*

[VVC18b] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *La programmation de microcontrôleurs dans des langages de haut niveau - Cours invité*. Dans *Actes des 29èmes Journées Francophones des Langages Applicatifs (JFLA 2018)*, BANYULS, France, janvier 2018. https://hal.sorbonne-universite.fr/hal-01762414. *[cited on page 208]*

[VVF18] Nicolas Valot, Pierre Vidal et Louis Fabre : *Increase avionics software development productivity using Micropython and Jupyter notebooks*. Dans *Proceedings of the 9th European Congress on Embedded Real Time Software and Systems (ERTS^2 2018)*, Toulouse, France, janvier 2018. https://hal.archives-ouvertes.fr/ERTS2018. *[cited on page 22]*

[VWC15] Benoît Vaugon, Philippe Wang et Emmanuel Chailloux : *Programming Microcontrollers in OCaml : The OCaPIC Project*. Dans *Proceedings of the 17th International Symposium on Practical Aspects of Declarative Languages (PADL 2015)*, tome 9131 de *Lecture Notes in Computer Science*, pages 132–148, Portland, OR, USA, juin 2015. Springer. https://doi.org/10.1007/978-3-319-19686-2_10. *[cited on page 23]*

[Wal00] David Walker : *A Type System for Expressive Security Policies*. Dans *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, pages 254–267, Boston, Massachusetts, USA, janvier 2000. ACM. https://doi.org/10.1145/325694.325728. *[cited on page 92]*

[WDS+10] W. E. Wong, V. Debroy, A. Surampudi, H. Kim et M. F. Siok : *Recent Catastrophic Accidents : Investigating How Software was Responsible*. Dans *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 14–22, juin 2010. https://doi.org/10.1109/SSIRI.2010.38. *[cited on page 34]*

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat et Per Stenström : *The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools*. ACM Transactions on Embedded Computing Systems (TECS), 7(3) :36 :1–36 :53, mai 2008, ISSN 1539-9087. https://doi.org/10.1145/1347375.1347389. *[2 citations pages 35 et 36]*

Abstract

Microcontrollers are programmable integrated circuit embedded in multiple everyday objects. Due to their scarce resources, they often are programmed using low-level languages such as C or assembly languages. These languages don't provide the same abstractions and guarantees than higher-level programming languages, such as OCaml. This thesis offers a set of solutions aimed at extending microcontrollers programming with high-level programming paradigms. These solutions provide multiple abstraction layers which, in particular, enable the development of portable programs, free from the specifics of the hardware. We thus introduce a layer of hardware abstraction through an OCaml virtual machine, that enjoys the multiple benefits of the language, while keeping a low memory footprint. We then extend the OCaml language with a synchronous programming model inspired from the Lustre dataflow language, which offers abstraction over the concurrent aspects of a program. The language is then formally specified and various typing properties are proven. Moreover, the abstractions offered by our work induce portability of some static analyses that can be done over the bytecode of programs. We thus propose such an analysis that consists of estimating the worst case execution time (WCET) of a synchronous program. All the propositions of this thesis form a complete development toolchain, and several practical examples that illustrate the benefits of the given solutions are thus provided.