PUC–Rio

Research Plan

Synchronous Languages for General Purpose Reactive Systems

Autor:
Francisco Sant'Anna

Advisors:
Roberto Ierusalimschy
Noemi Rodriguez

October 18, 2013

Contents

1 Abstract

The proposed research plan, entitled "Synchronous Languages for General Purpose Reactive Systems", aims to expand the use of static and safety-oriented synchronous languages to embrace general purpose reactive systems, such as GUIs, games, multimedia, and networked applications.

Synchronous languages typically target critical embedded systems, relying on a restricted semantics to enable static analysis in programs and provide a number of compile-time safety guarantees. On the one hand, synchronous programs are predictable in terms of memory usage, execution time, and behavior. On the other hand, they are limited in expressiveness, restricting support for dynamic memory allocation and for describing algorithmic-intensive computations. Although these limitations are not impeditive in the context of embedded systems, they are unacceptable for general purpose reactive programming.

Our goal is to study how to tune the safety and expressiveness boundaries of synchronous languages in a way that reactive applications can benefit of powerful mechanisms, while retaining as much efficiency and safety guarantees as possible.

2 Introduction

Concurrent languages can be generically classified in two major execution models. In the *asynchronous model*, the program activities (e.g. threads and processes) run independently of one another as result of non-deterministic preemptive scheduling. In order to coordinate at specific points, these activities require explicit use of synchronization primitives (e.g. mutual exclusion and message passing). In the *synchronous model*, the program activities (e.g. callbacks and coroutines) require explicit control/scheduling primitives (e.g. returning or yielding). For this reason, they are inherently synchronized, as the programmer himself specifies how they execute and transfer control.

Due to the simpler and disciplined execution model, synchronous languages are susceptible to static analysis and verification, being an established alternative to C in the field of safety-critical embedded systems [3]. However, in order to enable static analysis, synchronous programs typically suffer from two main limitations in expressiveness: *lack of safe dynamic memory allocation support* and *impossibility of doing algorithmic-intensive computations.* Memory allocation makes the static prediction about memory usage more difficult, transferring the burden of memory management to runtime and to programmers, which makes programs less safe. It also requires programmers to handle pointers explicitly, raising a number of safety threats, such as *dangling pointers* and *memory leaks*. Algorithm-intensive computations (e.g., compression, image processing) break the synchronous execution hypothesis, as control transfer among activities now happens in an unbounded amount of processing time [9]. For reactive applications, instantaneous feedback is fundamental, and minimal delays in responsiveness may affect the system correctness.

CÉU [15, 11, 13, 12] and LUAGRAVITY [14, 10] are two synchronous languages with distinct goals that we have been developing during the past 6 years.

CÉU is a Esterel-based [5] reactive language that targets constrained embedded platforms, such as Arduino[1] and wireless sensor nodes (e.g. *micaz*[2]). It is designed from scratch and pursues safety and efficiency. Most of the complexity of the language resides on the static analysis that happens at compile time. The compiler forbids unbounded loops and verifies that no two accesses to the same variable occur at the same time. It also calculates exact upper limits for memory usage, concurrent lines of execution, and nested invocations of events. Therefore, there are no runtime errors regarding memory allocations, race conditions, and unbounded execution.

LUAGRAVITY provides runtime extensions to the programming language Lua [7] for a better support for building reactive applications. LUAGRAVITY supports the emerging functional reactive programming style (FRP) for building declarative GUIs [8, 6, 2]. Being an extension module, instead of a new language on its own, LUAGRAVITY inherits all functionality of Lua, with unrestricted support for loops, recursive calls, and heap-allocated tables, closures, and coroutines. Therefore, LUAGRAVITY pursues power and flexibility, relying on a dynamic language that already pushes most safety checks to runtime.

Overall, CÉU produces more tractable and reliable programs, but also rapidly hits a limit in expressiveness when used outside its restricted embedded domain. Our goal with this research plan is to push the expressiveness limits of CÉU towards general purpose reactive systems. Starting from CÉU's restricted and tractable semantics, we can keep the precise memory and execution model that allows the compiler to predict resource usage. In contrast, most FRP systems expose turing-complete functional languages to programmers, and hence, cannot provide static safety guarantees.

[1] `http://www.arduino.cc`

[2] `http://www.xbow.com`

3 Research plan

Our goal is to extend the expressiveness of the programming language Céu, while accommodating two conflicting requirements:

- Keep the safety and resource efficiency of synchronous languages.
- Overcome the two main limitations of synchronous languages: *dynamic memory allocation* and *algorithmic-intensive computations.*

Céu was designed for constrained embedded systems, and we do not want to abdicate of its small footprint and compile-time safety guarantees. In order to overcome the limitations in expressiveness and embrace general-purpose reactive systems, we plan to add new functionality, keeping the safety mindset of the language:

- For dynamic allocation, we want to employ the use of memory pools under the hoods for predicting memory usage at compile time and enforcing deterministic behavior for memory management. We also want to take advantage of Céu's high level compositions to limit the visibility of allocated objects and reclaim memory efficiently through (static) lexical analysis.
- For algorithmic-intensive computations we plan to add an asynchronous primitive that cannot share memory with the reactive/synchronous code. With this approach, the reactive part of the program remains with the static guarantees, while the asynchronous execution has no restrictions, but cannot interfere with the synchronous execution.

After introducing Céu, we provide more details regarding the two proposals above and present possible application scenarios to evaluate the results.

3.1 The programming language Céu

Céu [13, 11, 12, 15] is a synchronous reactive language that supports multiple lines of execution—known as *trails*—that continuously react to external input events broadcast by the environment. The example in Figure 1 blinks three LEDs in parallel with different frequencies.

By following the synchronous model, each trail in Céu reacts to an occurring event with a *run-to-completion* policy: the scheduler of Céu is non-preemptive and deterministic. Reactions to subsequent events never overlap, and the compiler ensures that programs do not contain *unbounded loops* (i.e.,

```
par do
    loop do
        await 250ms;
        _led_toggle(0);
    end
with
    loop do
        await 500ms;
        _led_toggle(1);
    end
with
    loop do
        await 1000ms;
        _led_toggle(2);
    end
end
```

Figure 1: An introductory example in Céu that blinks three LEDs (identified as 0, 1, and 2) in parallel trails. The LEDs blink every $250ms$, $500ms$, and $1000ms$, respectively.

loops without `await` statements), which could lead to unresponsiveness for incoming events.

Programs in Céu are structured with standard imperative primitives, such as sequences, loops, and assignments. The extra support for parallelism allows programs to wait for multiple events at the same time. Trails await events without loosing context information, such as locals and the program counter, which eases reasoning in concurrent applications [1]. The conjunction of parallelism with standard control-flow enable hierarchical compositions, in which self-contained blocks of code can be deployed independently. To illustrate the power of compositions in Céu, consider the two variations of the structure that follows:

```
loop do
    par/and do
        <...>
    with
        await 1s;
    end
end
```

```
loop do
    par/or do
        <...>
    with
        await 1s;
    end
end
```

In the `par/and` loop variation, the code in the first trail (represented as ...) is repeated every second at minimum, as the second trail must also terminate to rejoin the `par/and` primitive and restart the loop. In the `par/or` loop variation, if the code does not terminate within a second, the second trail rejoins the composition (cancelling the first trail) and restarts the loop. The two structures represent, respectively, the *sampling* and *timeout* patterns, which are common in embedded applications. Note that the body of `<...>` may contain arbitrary code with nested compositions and awaits, but the described patterns will work as expected.

The use of parallel compositions under the disciplined synchronous execution model provides precise information about the control flow of applications, such as which lines of executions can be active at the same time, and which variables can be accessed in reaction to a specific event. As a summary, the following safety properties hold for all programs that successfully compile in CÉU [13]:

- Time-bounded reactions to the environment.
- No concurrency in accesses to shared variables.
- No concurrency in system calls sharing a resource.
- Synchronization for timers in parallel.

3.2 Dynamic memory allocation

Dynamic memory is a fundamental mechanism in general-purpose programming, allowing programs to accommodate short-lived objects without keeping them in memory for the whole execution.

CÉU has no special support for memory allocation, relying on standard *malloc* & *free* from *C*. However, standard memory management raises a number of challenges to the design of safe applications:

1. **Run-time overhead:** Memory management requires extra run-time bookkeeping for the allocation/reclamation algorithms. In real-time reactive applications, such as video games, even a small delay may be noticed by the user.

2. **Unreproducible execution:** Successive executions of the same program may allocate memory in different ways, possibly leading to different outcomes (e.g. an allocation fail).

3. **Reclamation hazards:** Properly reclaiming memory is far from trivial. A missed reclamation leads to a memory leak that wastes memory, while reclaiming a memory block still in use leads to a dangling pointer that will eventually crash the application.

A common alternative in the context of embedded systems is to use *memory pools* as the underlying scheme to manage dynamic allocation. A memory

```
1  par/or do
2      container do
3          // Self—contained block of code that uses
4          // dynamic allocation.
5          <...>
6              ref = alloc(<...>);
7          <...>
8      end
9  with
10     <...>
11 end
```

Figure 2: All memory allocated inside the `container` block is automatically reclaimed after it goes out of scope.

pool is a static buffer of N predefined fixed-sized blocks of memory that can be used by an application. With memory pools two of the threats raised above can be eliminated:

1. The run-time overhead is minimal because implementations typically use simple arrays to hold the memory blocks: allocation and reclamation are *O(1)*.

2. Given that memory operations are simple and only handle fixed-size blocks, the execution is always deterministic and predictable.

3. Memory pools are still manipulated through explicit allocation/reclamation operations. Hence, all challenges to properly reclaim memory still hold.

Regarding reclamation hazards, a typical alternative is to employ automatic memory management relying on a garbage collector. However, garbage collection increases the complexity of the language and also incurs a considerable runtime overhead.

The support for compositions and self-contained blocks of code in Céu opens an opportunity for an alternative approach that is efficient and avoids explicit memory reclamation by the programmer. In the example of Figure 2, the hypothetical `container` keyword (line 2) encloses a block of code that allocates dynamic memory multiple times (line 6). When the container goes out of scope (suppose the block in line 10 terminates), all memory allocated inside it can be automatically reclaimed. It is important that references to allocated blocks do not escape the container, which requires a trivial lexical analysis at compile time.

```
1  par/or do
2      <...> // defines ret, p1, p2
3      ret = async (p1, p2) do
4                <...>     // heavy computation
5                return v;
6            end
7  with
8      <...>
9  end
```

Figure 3: An `async` block executes independently ("detached") from the synchronous part of the program.

3.3 Algorithmic-intensive computations

Applications often need to perform some kind of heavy computation, such as compressing a file, or converting an image format. It is important that while the computation executes, the application does not "freeze" and remains reactive to external events. However, the synchronous execution model relies on the hypothesis that computations run infinitely faster than the rate of external events, which is not valid when facing heavy computations.

We agree on the fact that heavy computations are fundamentally incompatible with the synchronous model and propose the addition of an *asynchronous* primitive to CÉU. Such primitive has already being adopted by other synchronous systems [4, 6]. Figure 3 illustrates how an asynchronous execution can be started from the program. The `async` block (lines 3-6) may contain arbitrary code with heavy computations (line 4). On startup, the program may pass parameters by copy to the `async` (e.g. `p1` and `p2` in line 3). On termination, the `async` may return a value to the program (`return` in line 5 to the assignment in line 3). While the trail that spawned the `async` waits for its termination, other parts of the program in parallel (line 8) remain reactive.

An explicit boundary between synchronous and asynchronous execution make their uses clear and brings the benefits of both worlds:

Synchronous Safe and deterministic execution; Support for shared-memory concurrency; No need for extra synchronization primitives (e.g. *mutexes*); Reactive behavior with highest priority.

Asynchronous Non-deterministic execution; Share nothing concurrency; Parallelizable in multi cores; Background computations with lower priority.

3.4 Application scenarios

In order to evaluate the applicability of synchronous languages to general-purpose reactive systems, we intend to develop real applications in the scenarios that follow:

Games. Video games push the limits of reactive systems, requiring real time responsivenes for user interaction. Games also have plenty of heavy computations that allow us to explore asynchronous execution, such as collision detection and path finding algorithms.

Multimedia. Multimedia applications have a high degree of spatio and temporal synchronization among objects. There are plans to port Ginga [17] (the Brazilian middleware for Digital TV) to a pure synchronous implementation [16].

Network. Networked applications introduce an unreliable component in applications, which have to deal with communication delays and failures. We have already done a few experiments in Céu with undergraduate students in classes of distributed systems.

By developing applications in the described scenarios, we can also evaluate our design through quantitative metrics, as follows:

CPU usage. Both proposals of extensions to Céu can be evaluated through CPU usage: performance of automatic memory management and parallelization of asynchronous blocks of code.

Source code size. Even though source code size is not the ultimate metrics for code complexity, it provides straightforward means to compare different implementations for the same application.

4 Conclusion

General-purpose reactive systems can benefit from the efficiency and safety guarantees of synchronous languages. However, synchronous languages suffer from two main limitations:

- Lack of safe dynamic memory allocation support.
- Impossibility of doing algorithmic-intensive computations.

In this research plan, we proposed concrete directions to address these limitations:

- Employ the use of memory pools to predict memory usage and enforce deterministic behavior; Limit the scope of objects for automatic and efficient reclaiming of memory.

- Introduce a “share nothing” asynchronous primitive with no restrictions regarding heavy computations.

References

[1] Adya, A., et al. Cooperative task management without manual stack management. In *ATEC'02* (2002), USENIX Association, pp. 289–302.

[2] Bainomugisha, E., et al. A survey on reactive programming. *ACM Computing Surveys* (2012).

[3] Benveniste, A., et al. The synchronous languages twelve years later. In *Proceedings of the IEEE* (Jan 2003), vol. 91, pp. 64–83.

[4] Berry, G. *The Esterel-V5 Language Primer.* CMA and Inria, Sophia-Antipolis, France, June 2000. Version 5.10, Release 2.0.

[5] Boussinot, F., and de Simone, R. The Esterel language. *Proceedings of the IEEE 79*, 9 (Sep 1991), 1293–1304.

[6] Czaplicki, E., and Chong, S. Asynchronous functional reactive programming for guis. In *PLDI'13* (2013), pp. 411–422.

[7] Ierusalimschy, R. *Programming in Lua, Second Edition.* Lua.Org, 2006.

[8] Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices* (2009), vol. 44, ACM, pp. 1–20.

[9] Potop-Butucaru, D., et al. The synchronous hypothesis and synchronous languages. In *Embedded Systems Handbook*, R. Zurawski, Ed. 2005.

[10] Sant'Anna, F. A synchronous reactive language based on implicit invocation. Master's thesis, PUC–Rio, March 2009.

[11] Sant'Anna, F. *Safe System-level Concurrency on Resource-Constrained Nodes with Céu.* PhD thesis, PUC–Rio, 2013.

[12] Sant'Anna, F., et al. Advanced control reactivity for embedded systems. In *Proceedings of REM'13* (2013), ACM. to appear.

[13] Sant'Anna, F., et al. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of SenSys'13* (2013), ACM. to appear.

[14] Sant'Anna, F., and Ierusalimschy, R. LuaGravity, a reactive language based on implicit invocation. In *Proceedings of SBLP'09* (2009), pp. 89–102.

[15] Sant'Anna, F., Rodriguez, N., and Ierusalimschy, R. Céu: Embedded, Safe, and Reactive Programming. Tech. Rep. 12/12, PUC-Rio, 2012.

[16] Soares, L., et al. Revisiting the inter and intra media syncrhonization model of the ncl player architecture. In *Proceedings of Media Synchronization Workshop (MediaSync'13)* (2013). to appear.

[17] Soares, L. F. G., and Rodrigues, R. F. Nested Context Language 3.0 part 8 - NCL digital TV profiles. Tech. Rep. 35, Departamento de Informática - PUC-Rio, october 2006.