PUC–Rio

Research Plan

Synchronous Languages for General Purpose Reactive Systems

Autor:
Francisco Sant'Anna

Advisors:
Roberto Ierusalimschy
Noemi Rodriguez

October 16, 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 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 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*. Furthermore, it affects the performance of programs, given that memory allocation algorithms have some (possibly unpredictable) runtime overhead. 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 [8]. For reactive applications, instantaneous feedback is fundamental, and minimal delays in responsiveness may affect the system correctness.

Céu [14, 10, 12, 11] and LuaGravity [13, 9] are two synchronous languages with distinct goals that we have been developing during the past 6 years.

Céu is a Esterel-based [4] reactive language that targets constrained embedded platforms. It is designed from scratch and pursues safety and efficiency, targeting highly constrained embedded systems. 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 internal events. Therefore, there are no runtime errors regarding memory allocations, race conditions, and unbounded execution.

LuaGravity provides runtime extensions to the programming language Lua [6] for a better support for building reactive applications. LuaGravity supports the emerging functional reactive programming style (FRP) for building declarative GUIs [7, 5, 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 that 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. We believe that 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.

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

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

3 Research plan

3.1 The programming language Céu

Céu [12, 10, 11, 14] 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. Symbols defined externally in C[1], such as `_toggle` in the example, can be used seamlessly in Céu by prefixing their names with an underscore.

By following the synchronous model, each trail in Céu reacts to an occurring event with a *run-to-completion* policy and in the order they appear in the code: 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, which could lead to unresponsiveness for incoming events. The language runtime relies on a queue to hold incoming events while reactions execute to completion. The disciplined execution model of Céu simplifies the automatic analysis about concurrency aspects, enabling a number of static safety guarantees, such as race-free shared-memory and C calls [12].

Programs in Céu are structured with standard imperative primitives, such as sequences, loops, and assignments. The extra support for parallelism

[1]Symbols for functions, types, globals, and constants available in the C compiler for the target embedded platform.

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 conjunction of parallel compositions with the synchronous execution model provides precise information about the control flow of applications, such as which lines of executions can be active, which variables are accessed in reaction to a specific event, and so on. As a summary, the following safety properties hold for all programs that successfully compile in Céu [12]:

- 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

- memory pools
- compositions / scoped

3.3 Algorithmic-intensive computations

asynchoronous primitive

- share nothing policy
- lower priority
- automatic start/stop/schedule

3.4 General-purpose applications

- games
- multimedia (NCL, Telemidia)
- network (DS, Noemi)

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

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

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

[7] 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.

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

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

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

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

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

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

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