Imperative Reactive Programming with CÉU

April 17, 2014

The origins of reactive programming date back to the early 80's with the codevelopment of two styles of synchronous languages: The imperative style of Esterel specifies programs with control flow primitives, such as sequences, repetitions, and also parallelism. The dataflow style of Lustre represents programs as graphs of values, in which a change to a node updates its dependencies automatically.

In recent years, Functional Reactive Programming modernized the dataflow style an became mainstream, producing a number of languages and libraries, such as Flapjax, Rx (from Microsoft), React (from Facebook), and Elm. In contrast, the imperative style did not follow this trend and is now confined to the domain of real-time embedded control systems.

We present the programming language CÉU, a contemporary outlook of imperative reactivity, aiming to expand the application domain of this family with a new abstraction mechanism. CÉU has its roots on Esterel and relies on a similar synchronous and deterministic execution model that simplifies the reasoning about concurrency aspects. The example below in CÉU prints the "Hello World!" message every second, terminating with a key press:

```
par/or do
   every 1s do
      print("Hello World!");
   end
with
   await KEY;
end
```

A `par/or` composition spawns concurrent trails (CÉU's lightweight lines of execution) that rejoin when any of them terminate (an analogous `par/and` rejoins when all trails terminate). The first trail continuously prints "Hello World!", while the second awaits for a key press to terminate the whole composition.

The `await` statement is the most representative of CÉU, capturing the imperative and reactive nature of the language at the same time. Awaiting multiple events in parallel compositions releases programmers from the infamous *callback hell*, which is often a concern in general purpose imperative languages.

CÉU introduces a new class-based abstraction mechanism that includes an execution body with reactive statements. The example below defines a class "Hello" that prints a custom message every second. In the program body, we declare two instances inside a `do-end` block that limits their scope:

```
class Hello with
   var int i;
do
   every 1s do
      print("I am %d!", this.i);
   end
end

do
   var Hello a with
      i = 1;
```

```
    end;
    var Hello b with
       i = 2;
    end;
    await KEY;
 end
```

When the program starts, each instance spawns its body in parallel with the `do-end` block that will await the key press. On user input, the instances go out of scope with the enclosing block and their bodies are automatically terminated. In CÉU, instances have lexical scope just like local primitive variables, eliminating the overhead of a garbage collector. An static analysis ensures that references to instances never leak to outer scopes.

Even though other languages support local instances (e.g. C++), the presence of the `await` statement makes possible for blocks in CÉU to outlive multiple external reactions while keeping locals alive. Even dynamically allocated instances have lexical scope in CÉU, as illustrated below:

```
par/or do
   do
      every 1s do
         spawn Hello with
            i = _rand();
         end;
      end
   end
with
   await KEY;
end
```

We now use the `spawn` primitive to create a new instance of `Hello` every second. The dynamic instances are bound the enclosing `do-end` block. On user input, the whole `par/or` terminates, deallocating all instances in the block. Note how allocated instances are conveniently anonymous, releasing the programmer from managing their life cycles.

To evaluate performance, we wrote a stress test program that spawns 10,000 animated graphical objects, each containing 3 trails (resulting in 30,000 coexisting trails). We compared it with a pure event-driven implementation in C and noticed an acceptable decrease of 10-20% in the frame rate for the implementation in CÉU.

We believe that the imperative reactive style of CÉU is a realistic counterpart to Functional Reactive Programming and complements the realm of synchronous programming. The new abstraction mechanism of CÉU makes imperative reactivity suitable for applications outside the embedded domain, expanding classical object orientation with instances that are reactive to the environment. As an example, we implemented in CÉU an Android game of moderate complexity (approximately 5,000 lines of code).