An Overview of Céu

Francisco Sant'Anna

`francisco@ime.uerj.br`

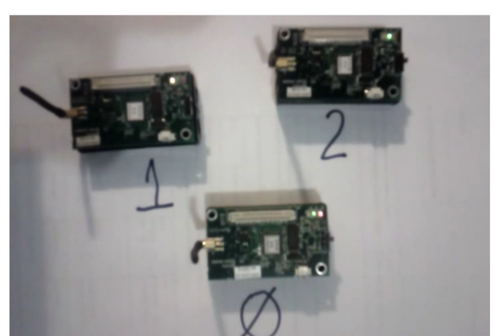
www.ceu-lang.org

Céu goals

- Front-end applications
    - GUIs, Games, interactive apps
    - Desktops → Arduino *(4K RAM)*
- Ruled by the environment
    - Immediate/real-time feedback
    - Global consensus
- Logical reasoning
    - Concurrency w/ Determinism

Céu non-goals

- Back-end infrastructure
    - High-performance servers
    - Clusters, Cloud computing
- Independent sessions/actors
    - Latency in communication
    - Distribution
- The C10M problem
    - Concurrency w/ Parallelism



“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*
  3. *restart after 2s*

```
loop do
    par/or do
        loop do
            await 500ms;
            _leds_toggle();
        end
    with
        await PRESS;
    end
    await 2s;
end
```

Lines of execution
=
Trails (in Céu)

- Compositions
  - seq, loop, par *(trails)*
    - At any level of depth
  - ~~state variables / communication~~

From "Structured Programming"
To "Structured Reactive Programming"

- Control Structures
  - Sequences, Loops, Conditionals
- Blocks, Scopes, Locals
  - Lexical memory management
- Subroutines
  - Abstraction mechanism
- What about reactivity?
  - Environment event → Short-lived callback
    - No more loops, scopes, etc.
    - Breaks structured programming
      - "Callbacks as our Generations' `goto`" [Miguel de Icaza]

- The `await` statement
  - *Imperative-reactive* nature
- Compositions
  - Control structures + parallels
- Synchronous execution model
  - Time ~ Sequence of events
- Esterel did this back in the *'80s*
- What about abstractions and modularization?

Two Blinking LEDs

[led-04]
(copy&paste)

```
loop do
   par/
      l

      e
   with
      a
   end
   await 2s;
end
```

```
par do
   loop do
      par/or do
         every 500ms do
            _leds_toggle(1);
         end
                                          PRESS
                                          bt==1;

      par/or do
         every 500ms do
            _leds_toggle(2);
         end
      with
         var int bt = await PRESS
                      until bt==2;
      end
      await 2s;
   end
end
```

The need for abstractions!

Céu Abstractions

- `code/await`: a procedure that can `await`

```
code/await Led (var int led, var int but) -> FOREVER
do
    loop do
        par/or do
            every 500ms do
                _leds_toggle(led);
            end
        with
            var int bt = await PRESS
                             until bt==but;
        end
        await 2s;
    end
end
```

A code/await reacts directly to the environment

[led-05]
(code/await)

Two Blinking LEDs

```
code/await Led (var int led, var int but) -> FOREVER
do
    <...>
end

par do
    await Led(1,1);
with
    await Led(2,2);
end
```

LEDs are reacting in parallel

await invokes a code/await and waits for its termination

6-Blinking LEDs

```
code/await Led (var int led, var int but) -> FOREVER
do
    <...>
end

par do
    await Led(1,1);
with
    await Led(2,2);
with
    await Led(3,3);
with
    await Led(4,4);
with
    await Led(5,5);
with
    await Led(6,6);
end
```

Lexical Scope

```
code/await Led (var int led, var int but) -> FOREVER
do
    <...>
end


loop do
    par/or do
        var int bt = await PRESS until bt==0;
    with
        await Led(1,1);
    with
        await Led(2,2);
    with
        await Led(3,3);
    with
        await Led(4,4);
    with
        await Led(5,5);
    with
        await Led(6,6);
    end
end
```

`code/await` has lexical scope

`code/await` out of scope: data reclaimed **and body aborted**

`code/await` management is as simple as local variables

Structured, Reactive,
Abstract.

But we still have a static semantics!

6-Blinking LEDs with Lexical Scope

```
loop do
    par/or do
        loop do
            await 500ms;
            _leds_toggle(i);
        end
    with
        await i-PRESS;
    end
    await 2s;
end
```

```
loop do
    par/or do
        par do
            <body-1>
        with
            ...
        with
            <body-n>
        end
    with
        var int bt =
            await PRESS
            until bt==0;
    end
end
```

Another Example

[bird-01]
(code/await)

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    <...>
end
```

Reaction to the environment is abstracted inside the body

```
par do
    await Bird(100,100);
with
    await Bird(300,200);
end
```

On instantiation, only the interface matters

[bird-02]
(scope)

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    <...>
end

loop do
    par/or do
        await Bird(100,100);
    with
        await Bird(100,100);
    with
        await MOUSE_CLICK;
    end
end
```

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    <...>
end

pool[5] Bird birds;

loop i in [1 -> 5] do
    spawn Bird(70*i, 100+10*i) in birds;
end

await FOREVER;
```

A pool is a container for code/await instances

spawn invokes a code/await and continues

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    <...>
end

pool[5] Bird birds;

every 1s do
    spawn Bird(20+_rand()%HEIGHT, 100+_rand()%100) in birds;
end
```

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    <...>
end

pool[] Bird birds;

every 1s do
    spawn Bird(...) in birds;
end
```

[bird-05]
(Bird implementation)

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    par do
        every UPDATE do
            <animate-bird>
        end
    with
        every DRAW do
            <draw-bird>
        end
    end
end

pool[] Bird birds;

every 1s do
    spawn Bird(...) in birds;
end
```

animation trail

drawing trail

[bird-06]
(break/unbounded)

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    par do
        every UPDATE do
            if x >= WIDTH then
                break;
            end
            <animate-bird>
        end
    with
        every DRAW do
            <draw-bird>
        end
    end
end

pool[] Bird birds;

every 1s do
    spawn Bird(...) in birds;
end
```

animation trail

drawing trail

[bird-07]
(break/bounded)

```
code/await Bird (var int y, var int speed) -> FOREVER
do
    par do
        every UPDATE do
            if x >= WIDTH then
                break;
            end
            <animate-bird>
        end
    with
        every DRAW do
            <draw-bird>
        end
    end
end

pool[5] Bird birds;

every 1s do
    spawn Bird(...) in birds;
end
```

animation trail

drawing trail

[bird-08]
(natural termination)

```
code/await Bird (var int y, var int speed) -> void
do
    par/or do
        every UPDATE do
            <animate-bird/break>
        end
    with
        every DRAW do
            <draw-bird>
        end
    end
end

pool[5] Bird birds;

every 1s do
    spawn Bird(...) in birds;
end;
```

animation trail

drawing trail

the whole body terminates (natural termination)

natural termination automatically reclaims dynamic `code/await` instances from memory pools

no need for a free primitive or garbage collection

spawned **`code/await`** is anonymous

```
code/await Bird (var int y, var int speed) -> void
do
    <...>
end

loop do
    par/or do
        pool Birds[] birds;
        every 1s do
            spawn Bird(...) in birds;
        end
    with
        await MOUSE_CLICK;
    end
end
```

pools also have lexical scope

`pool` out of scope: data and execution body of all alive **`code/await`** instances are reclaimed

heap allocation with lexical memory management (vs. garbage collection)

```
code/await Bird (var int y, var int speed) -> void
do
    <...>
end

loop do
    watching MOUSE_CLICK do
        pool Birds[] birds;
        every 1s do
            spawn Bird(...) in birds;
        end
    end
end
```

[birds-10]
(collision/iterator)

```
loop do
    watching MOUSE_CLICK do
        pool Birds[] birds;
        par do
            every 1s do
                spawn Bird(...) in birds;
            end
        with
            every UPDATE do
                var&   Rect rct1;
                event& void col1;
                loop (rct1,col1) in birds do
                    var&   Rect rct2;
                    event& void col2;
                    loop (rct2,col2) in birds do
                        if (&rct1<&rct2) and Int(&rct1,&rct2) then
                            emit col1;
                            emit col2;
                            break;
                        end
                    end
                end
            end
        end
    end
end
```

temporary alias
(valid only inside the `loop`)

iterate over the birds

access to internal state
of a `code/await` in the pool

notifies both `code/await`
about the collision

```
code/await Bird (var int y, var int speed)
                      -> (var& Rect rct, event& void collide)
                         -> FOREVER
do
    var Rect my_rct = val Rect(20,y, 50,45);
    rct = &my_rct;

    event void my_collide;
    collide = &my_collide;

    par/or do
        watching my_collide do
            every UPDATE do
                <animate/break> // accesses my_rct
            end
        end
    with
        every DRAW do
            <draw-bird>          // accesses my_rct
        end
    end
end
```

a `code/await` may expose
and share its internal state

```
code/await Bird (...) -> (...) -> void
do
    <...>
    par/or do
        watching my_collide do
            every UPDATE do
                <animate-bird/break>
            end
        end
        every UPDATE do
            <animate-bird-y>
            if my_rct.y >= HEIGHT then
                break;
            end
        end
    with
        every DRAW do
            <draw-bird>
        end
    end
end
```

[birds-12]
(blink / pause/if)

```
code/await Bird (...) -> (...) -> void
do
    <...>
    event bool hide;
    par/or do
        watching my_collide do
            every UPDATE do
                <animate-bird/break>
            end
        end
        every UPDATE do
            <animate-bird-fall>
        end
        watching 1s do
            loop do
                emit hide(true);
                await 100ms;
                emit hide(false);
                await 100ms;
            end
        end
    with
        pause/if hide do
            every DRAW do
                <draw-bird>
            end
        end
    end
end
```

```
event bool freeze;
par do
    pause/if freeze do
        <...> // bird pool, creation and collision
    end
with
    loop do
        var Key key1 = await KEY_PRESS until key1.sym=='p';
        emit freeze(true);
        par/or do
            every DRAW do
                <draw-pause>
            end
        with
            var Key key2 = await KEY_PRESS until key2.sym=='p';
        end
        emit freeze(false);
    end
end
```

[birds-14]
(line)

```
pool Birds[] birds;
par do
    every 1s do
        spawn Bird(...) in birds;
    end
with
    every UPDATE do
        <collision-detection>
    end
with
    loop do
        var Mouse mse = await MOUSE_CLICK;
        var&? Rect found = do
            var&? Rect rct;
            loop (rct,_) in birds do
                if Rect_vs_Mouse(&rct,mse) then
                    escape &rct;
                end
            end
        end;
        watching found do
            every DRAW do
                <draw-line>
            end
        end
    end
end
```

checks if a bird was clicked

watches the bird while drawing the line

Summary

- Structured Programming ->
  Structured Synchronous Reactive Programming
    - sequences, loops, conditionals
        - await + parallels + abortion
    - procedures/abstractions
        - code/await
    - static/dynamic abstractions
        - lexical scope

An Overview of Céu

Francisco Sant'Anna

francisco@ime.uerj.br

LabLua
programming language research

ceu
www.ceu-lang.org

UERJ
UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO