Reactive Traversal of Recursive Data Types (?!)

Francisco Sant'Anna
Departamento de Informática — PUC-Rio, Brazil
fsantanna@inf.puc-rio.br

Hisham Muhammad
Departamento de Informática — PUC-Rio, Brazil
hisham@inf.puc-rio.br

Johnicholas Hines
Affiliation
email@domain.com

ABSTRACT
We propose a structured mechanism to traverse recursive data types incrementally, in successive reactions to input events. `traverse` is an iterator-like anonymous block that can be invoked recursively and suspended at any point, retaining the full state and stack frames alive. `traverse` is designed for the synchronous language Céu, inheriting all existing concurrency functionality, such as parallel compositions with orthogonal abortion, static memory management, and bounded memory and reaction time. We present two application scenarios that take advantage of recursive and reactive behavior: *incremental computation* and *control-oriented domain specific languages*.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms
Design, Languages

Keywords
Behavior Trees, Domain Specific Languages, Incremental Computation, Logo, Structured Programming

1. INTRODUCTION
...

... Céu [1, 2]

...

```
1  input void RESET;   // declares an external event
2  var int v = 0;      // variable shared by the trails
3  par do
4     loop do       // 1st trail
5        await 1s;
6        v = v + 1;
7        _printf("v = %d\n", v);
8     end
9  with
10    loop do       // 2nd trail
11       await RESET;
12       v = 0;
13    end
14 end
```

Figure 1: Introductory example in Céu.

2. CÉU
Céu is a concurrent and reactive language in which the lines of execution, known as *trails*, react all together continuously and in synchronous steps to external stimuli. The introductory example in Figure 1 defines an input event `RESET` (line 1), a shared variable `v` (line 2), and starts two trails with the `par` construct (lines 3-14): the first (lines 4-8) increments variable `v` on every second and prints its value on screen; the second (lines 10-13) resets `v` on every external request to `RESET`. Céu is tightly integrated with *C* and can access libraries of the underlying platform directly by prefixing symbols with an underscore (e.g., `_printf(<...>)`, in line 7).

In the synchronous model of Céu, a program reacts to an occurring event completely before handling the next. A reaction represents a logical instant in which all trails awaiting the occurring event awake and execute, one after the other, until they await again or terminate. During a reaction, the environment is invariant and does not interrupt the running trails[1]. If multiple trails react to the same event, the scheduler employs lexical order, i.e., the trail that appears first in the source code executes first. For this reason, programs are deterministic even in the presence of side effects in concurrent lines of execution. To avoid infinite execution for reactions, Céu ensures that all loops contain `await` statements [1].

2.1 Recursive Data Types

[1]The actual implementation enqueues incoming input events to process them in further reactions.

```
1  data List with
2     tag NIL;
3  or
4     tag CONS with
5        var int   head;
6        var List* tail;
7     end
8  end
9
10 do
11    pool List[1] l1;
12    l1 = new List.CONS(1,
13              List.CONS(2,
14               List.NIL()));
15    _printf("%d, %d\n", l1:CONS.head,
16                    l1:CONS.tail:NIL);
17       // prints 1, 1
18 end
19
20 do
21    pool List[] l2;
22    l2 = new List.CONS(1,
23              List.CONS(2,
24               List.NIL()));
25    l2:CONS.tail = new List.CONS(3, List.NIL());
26    _printf("%d\n", l2:CONS.tail:CONS.head);
27       // prints 3 (2 has been freed)
28 end
```

Figure 2: A recursive `List` data type definition (lines 1–8) and uses (lines 10–18 and 20–28).

CÉU supports algebraic data types (ADTs) as a safer alternative to C's `struct`, `union`, and `enum` definitions. However, CÉU preserves the mutable semantics of data types in C, and extends it with static memory management. For this reason, ADTs in CÉU differ in fundamental ways from functional algebraic data types (*a la* Haskell and ML). Currently, there is no parametric support for ADTs in CÉU.

Figure 2 illustrates the recursive `List` data type, which is either an empty list `NIL` (line 2) or a `CONS` with a value in the field `head` and a pointer to the rest of the list in the field `tail` (lines 4–7).

In the first block (lines 10–18), the `pool` declaration of `l1` represents the root of the list and also specifies a memory pool to hold all of its elements (line 10). We limit `l1` to contain at most 1 element (i.e., `List[1]`). The declaration also implicitly initializes the root to be the base case of the associated data type (i.e., `List.NIL`). Then, we mutate the root element to point to a dynamically allocated list of two elements (lines 12–14). The assignment infers the destination memory pool based on the *l-val* of the expression (i.e., `l1`). In this case, only the allocation of first element succeeds, with the failed allocation returning the base case of the data type (i.e., `List.NIL`). The print command (line 15) outputs *"1, 1"*: the `head` of the first element (the operator '`:`' is equivalent to C's '`->`') and the `NIL` check of the second element. Due to static memory management, when `l1` goes out of scope, at the end of the block (line 18), all elements in the list are automatically deallocated.

In the second block (lines 20–28), we declare the `l2` pool with an unbounded memory limit (i.e., `List[]` in line 21). Now, the two-element allocation succeeds (lines 22–24). Then, we mutate the tail of the first element to point to a newly allocated element, which also succeeds (line 25). The print command (line 26) outputs *"3"*, the new `head` of the second element. In the moment of the mutation, the old subtree is completely removed from memory. Finally, the end of the block (line 28) deallocates the pool along with all of its elements.

```
1  pool List[] l = new List.CONS(1,
2                       List.CONS(2,
3                        List.CONS(3,
4                         List.NIL()))));
5
6  var int sum =
7     traverse e in l do
8        if e:NIL then
9           escape 0;
10       else
11          var int sum_tail = traverse e:CONS.tail;
12          escape sum_tail + e:CONS.head;
13       end
14    end;
15 _printf("sum = %d\n", sum);
```

Figure 3: Calculating the *sum* of a list.

Data types in CÉU have a number of limitations: given that mutations deallocate whole subtrees, data types cannot represent general graphs (in particular, they cannot contain cycles); elements in different pools cannot be mixed; and pointers to subtrees (i.e., weak references) must be observed as they can be deallocated at any time (to be discussed in Section 2.2).

2.2 Traversing Data Types

CÉU introduces a structured mechanism to traverse data types. The `traverse` construct integrates well with the synchronous execution model, supporting nested control compositions, such as `await` and all `par` variations. It also preserves explicit lexical scopes with static memory management.

The example in Figure 3 creates a list (lines 1–4) and traverses it to calculate the sum of elements (lines 6–15). The `traverse` block (line 7) starts with the element `e` pointing to the root of the list `l`. The `escape` statement (lines 9 and 12) returns a value to be assigned to the `sum` (line 6). A `NIL` list has `sum=0` (lines 8-9). A `CONS` list needs to calculate the `sum` of its tail recursively, invoking `traverse` again (line 11), which will create a nested instance of the enclosing `traverse` block (lines 7–14), now with `e` pointing to the `e:CONS.tail`. Without nested control mechanisms, `traverse` is just syntactic sugar for anonymous closures called recursively.

To distinguish `traverse` from standard recursive functions, Figure 4 extends the body of the previous example with reactive behavior. For each recursive iteration, the `traverse` prints the current `head` (line 8) and awaits 1 second before traversing the `tail` (lines 9–10). In CÉU, all accesses to pointers that cross `await` statements must be protected with `watching` blocks [2]. This ensures that if side effects occurring in parallel affect the pointed object, no harming code executes because the whole block is aborted. In the example (lines 7–12), if the list is mutated during that 1 second and the specific element is removed from memory, we simply ignore the whole subtree and return 0.

```
pool List[] l = <...>;  // 1, 2, 3
var int sum =
   traverse e in l do
      if e:NIL then
         escape 0;
      else
         watching e do
            _printf("me = %d\n", e:CONS.head);
            await 1s;
            var int sum_tail = traverse e:CONS.tail;
            escape sum_tail + e:CONS.head;
         end
         escape 0;
      end
   end;
_printf("sum = %d\n", sum);
```

Figure 4: Calculating the *sum* of a list, one element each second.

Note that for bounded pools (e.g., `List[3] l`), we can infer at compile time the maximum number of "stack frames" required for `traverse`. In addition, we can also enforce bounded execution time by asserting that the structure of the recursive steps converge to the base cases. This is an important requirement for constrained and real-time embedded systems, which is the original application domain of CÉU [1].

3. APPLICATIONS

incremental computation, behavior trees, control-dominated DSLs

...

3.1 Incremental Computation

...

- gray binary generation?

...

3.2 Domain Specific Languages

3.2.1 Behavior Trees

The term "Behavior Trees" denotes a family of DSLs used for Game AI. The term is loose, because different games use different languages, but generally it indicates an interpreted domain-specific language for creature behavior that includes at least sequence and selection combinators, and which are "ticked" periodically.

The semantics of the sequence combinator can be understood as short-circuit evaluation of a conjunction; the Seq node ticks its left subtree until it finishes, and if it finishes successfully, ticks its right subtree until it finishes. The semantics of the selection combinator can be understood as short-circuit evaluation of an alternation; the Sel node ticks its left subtree until it finishes, and if it did not finish successfully, ticks its right subtree until it finishes.

This skeleton, augmented with leaves that test properties, set properties, perform animations and sounds, and other custom combinators, can be preferable to finite state machines (hierarchical, augmented, or otherwise) for authoring Game AI.

```
data Command with
    tag NOTHING;
or
    tag ROTATE with
        var int angle;
    end
or
    tag MOVE with
        var int pixels;
    end
or
    tag AWAIT with
        var int ms;
    end
or
    tag SEQUENCE with
        var Command* one;
        var Command* two;
    end
or
    tag REPEAT with
        var int      times;
        var Command* command;
    end
or
    tag PAROR with
        var Command* one;
        var Command* two;
    end
end
end
```

Figure 5: DSL for a LOGO turtle.

Ceu's parallel features make implementing a parallel combinator for behavior trees much easier.

...

3.2.2 Logo Turtle

...

4. RELATED WORK

...

5. CONCLUSION

...

6. REFERENCES

[1] F. Sant'Anna et al. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM, 2013.

[2] F. Sant'Anna et al. Structured Synchronous Reactive Programming with Céu. In *Proceedings of Modularity'15*, 2015. to appear.

```
class Interpreter with
    pool Command[]& cmds;
    var  Turtle&    turtle;
do
    traverse cmd in cmds do
        watching cmd do
            if cmd:AWAIT then
                await (cmd:AWAIT.ms) ms;

            else/if cmd:ROTATE then
                do TurtleRotate with
                    this.turtle = turtle;
                    this.angle  = cmd:ROTATE.angle;
                end;

            else/if cmd:MOVE then
                do TurtleMove with
                    this.turtle  = turtle;
                    this.pixels = cmd:MOVE.pixels;
                end;

            else/if cmd:PAROR then
                par/or do
                    traverse cmd:PAROR.one;
                with
                    traverse cmd:PAROR.two;
                end

            else/if cmd:SEQUENCE then
                traverse cmd:SEQUENCE.one;
                traverse cmd:SEQUENCE.two;

            else/if cmd:REPEAT then
                loop i in cmd:REPEAT.times do
                    traverse cmd:REPEAT.command;
                end
            end
        end
    end
end
```

Figure 6: The turtle interpreter.