# Traverse


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 structures incrementally. `traverse` is ...

MIX OF:

- recursive calls to anonymous closures
- each instance—many co-routines

DESIGNED FOR Céu:

- lexical compositions
- static memory management
- bounded execution/memory
- reactive
- mutation



## Categories and Subject Descriptors

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

## General Terms

Design, Languages

## Keywords

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


## 1. INTRODUCTION

...

... Céu [1, 2]

...

```
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
```

**Figure 1: A `List` data type with two constructors: `NIL` builds an empty list, and `CONS` builds an item `head` with a `tail` pointing to the rest of the list.**

## 2. CÉU

- adts
- description
- expansion: pool / recursive spawn
- mutation / safety / watching

### 2.1 Data Types

Céu supports algebraic data types as an alternative to `struct`, `union`, and `enum` declarations of C. For this reason, they differ from functional algebraic data types (*a la* Haskell and ML) and are mutable and have static memory management. Currently, there is no support for parametric types.

The `List` type of Figure 1 is either an empty list `NIL` (line 2) or a `CONS` with a value in the `head` and a pointer to the rest of the list in the `tail` (lines 4–7). The code in Figure 2 illustrates the use of the `List` data type.

In the first block (lines 1–9), the `pool` declaration of `l1` represents the root of the list and also specifies a memory pool to hold all of its elements (line 1). 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 3–5). 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 6) outputs *1, 1*: the `head` of the first element 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 9), all elements in the list are automatically deallocated.

```
1  do
2      pool List[1] l1;
3      l1 = new List.CONS(1,
4                List.CONS(2,
5                 List.NIL()));
6      _printf("%d, %d\n", l1:CONS.head,
7                          l1:CONS.tail:NIL);
8          // prints 1, 1
9  end
10 do
11     pool List[] l2;
12     l2 = new List.CONS(1,
13               List.CONS(2,
14                List.NIL()));
15     l2:CONS.tail = new List.CONS(3, List.NIL());
16     _printf("%d\n", l2:CONS.tail:CONS.head);
17         // prints 3 (2 has been freed)
18 end
```

**Figure 2: Declaration, allocation, mutation, and deallocation of `List` data types.**

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

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

In the second block (lines 10–18), we declare the `l2` pool with an unbounded memory limit (i.e., `List[]` in line 11). Now, the two-element allocation succeeds (lines 12–14). Then, we mutate the tail of the first element to point to a newly allocated element, which also succeeds (line 15). The print command (line 16) 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 18) deallocates the pool along with all of its elements.

Data types in CÉU has a number of limitations: given that mutations deallocate whole subtrees, data types cannot represent general graphs (in particular, cannot represent 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 (discussed in Section **??**).

## 2.2 Traverse

...

The code in Figure 3 creates a list (lines 1–5) and traverses it to calculate the sum of elements (lines 7–16).

...

# 3. APPLICATIONS

```
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 4: DSL for a LOGO turtle.**

...

## 3.1 Incremental Computation

...

- gray binary generation?

...

## 3.2 Behavior Trees

...

- ?

...

## 3.3 Domain Specific Languages

...

- 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 5: The turtle interpreter.**