

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

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]

...

2. CÉU

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

```
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.CONNS(1,
13     List.CONNS(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.CONNS(1,
23     List.CONNS(2,
24       List.NIL()));
25   l2:CONS.tail = new List.CONNS(3, List.NIL());
26   _printf("%d\n", l2:CONS.tail:CONS.head);
27   // prints 3 (2 has been freed)
28 end
```

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

2.1 Recursive Data Types

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 1 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

```

1 pool List[] l = new List.CONST(1,
2     List.CONST(2,
3     List.CONST(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 2: Calculating the *sum* of a list.

elements (lines 12–14). The assignment infers the destination memory pool based on the *l-val* of the expression (i.e., 11). 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 11 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 12 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.

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 2 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 1. 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`. With-

```

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

```

Figure 3: Calculating the *sum* of a list, one element each second.

out nested control mechanisms, `traverse` is just syntactic sugar for anonymous closures called recursively.

To distinguish `traverse` from standard recursive functions, Figure 3 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.

Note that for bounded pools (e.g., `List[3] 1`), 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 Behavior Trees

...

- ?

...

```

1 data Command with
2   tag NOTHING;
3 or
4   tag ROTATE with
5     var int angle;
6   end
7 or
8   tag MOVE with
9     var int pixels;
10  end
11 or
12   tag AWAIT with
13     var int ms;
14   end
15 or
16   tag SEQUENCE with
17     var Command* one;
18     var Command* two;
19   end
20 or
21   tag REPEAT with
22     var int times;
23     var Command* command;
24   end
25 or
26   tag PAROR with
27     var Command* one;
28     var Command* two;
29   end
30 end
31 end

```

Figure 4: DSL for a LOGO turtle.

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.

```

1 class Interpreter with
2   pool Command[]& cmds;
3   var Turtle& turtle;
4 do
5   traverse cmd in cmds do
6     watching cmd do
7       if cmd:AWAIT then
8         await (cmd:AWAIT.ms) ms;
9
10      else/if cmd:ROTATE then
11        do TurtleRotate with
12          this.turtle = turtle;
13          this.angle = cmd:ROTATE.angle;
14        end;
15
16      else/if cmd:MOVE then
17        do TurtleMove with
18          this.turtle = turtle;
19          this.pixels = cmd:MOVE.pixels;
20        end;
21
22      else/if cmd:PAROR then
23        par/or do
24          traverse cmd:PAROR.one;
25          with
26            traverse cmd:PAROR.two;
27          end
28
29      else/if cmd:SEQUENCE then
30        traverse cmd:SEQUENCE.one;
31        traverse cmd:SEQUENCE.two;
32
33      else/if cmd:REPEAT then
34        loop i in cmd:REPEAT.times do
35          traverse cmd:REPEAT.command;
36        end
37      end
38    end
39  end
40 end

```

Figure 5: The turtle interpreter.