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
IDEXX Laboratories
johnicholas.hines@gmail.com

ABSTRACT

We propose a structured mechanism to traverse recursive data types incrementally, in successive reactions to external 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 of its concurrency functionality and safety properties, such as parallel compositions with orthogonal abortion, static memory management, and bounded reaction time and memory usage. We discuss three applications in the domains of incremental computation and control-oriented DSLs that contain reactive and recursive behavior at the same time.

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, Recursive Data Types, Structured Programming, Reactive Programming

1. INTRODUCTION

The facilities a given language offers for constructing data types have a direct impact on the nature of algorithms that programmers will write on that language. As an example, the aim for referential transparency in functional languages enforces data structures to be immutable. Under these constraints, one must avoid excessive memory copying through specialized algorithms [6].

In this paper, we discuss the design of recursive data types and an associated control facility for a language developed under a different set of constraints. CÉU [8, 9] is an imperative, concurrent and reactive language in which the lines of execution, known as *trails*, react all together continuously and in synchronous steps to external stimuli. At the same time, it promotes mutation of data structures, static memory management through lexically-scoped memory pools, and safe pointer manipulation. These features are incompatible with garbage-collected immutable data structures, but also preclude the availability of general records with arbitrary pointers such as *structs* in C.

The solution to this problem is twofold, with data and control aspects. For data management, we introduce a restricted form of data types allowing mutation and static memory management. For handling reactive control, we propose a structured mechanism that can traverse recursive data types safely and incrementally, in successive reactions to input events.

After we present the design of these constructs in Section 2, we discuss three applications in the domains of incremental computation and control-oriented DSLs in Section 3. The applications contain reactive and recursive behavior at the same time and showcase the expressiveness of the proposed constructs. Then, we discuss some related work in Section 4 and make closing remarks in Section 5.

```
 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 CONSTRUCTS

In this section, we present the constructs added to CÉU to support recursive data types with reactive traversal.

The introductory example in Figure 1 gives a general flavor of the language. It first 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. 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 [8].

2.1 Recursive Data Types

The `data` construct in CÉU provides a safer alternative to C's `struct`, `union`, and `enum` definitions. Figure 2 illustrates the recursive `List` data type, declared as a tagged union (lines 1–5). The first tag, `NIL` (line 2), represents the empty list and is the union's null type. The second tag, `CONS` (line 4), receives two values in the constructor: the field `int head` and the rest of the list in the recursive field `List tail`.

In the first block of the example (lines 7–16), we declare a `pool` of `List` objects of size 1 (line 8). All recursive `data` instances must reside in an explicit memory pool, which have static memory management based on its lexical scope (lines 7–16). A pool also represents the reference to the root instance, which is implicitly initialized to the *null* tag of the associated data type, i.e., `lst1` receives `List.NIL` (line 8). Then, we use the `=new` construct (lines 9–12) which performs allocation and assignment at the same time: it attempts to dynamically allocate a list of three elements (10, 20, and 30), inferring the destination memory pool based on the assignment's *l-value* prefix (i.e. `lst1`). Since the pool has size 1, only the allocation of first element succeeds, with the failed subtree allocation returning the *null* tag (i.e., `List.NIL`). The print command (lines 13–14) outputs "`10, 1`": the `head` of the first element and a true value for the `NIL` check of the second element. Finally, the end of the block (line 16) deallocates the pool along with all elements inside it.

In the second block (lines 18–24), we declare the `lst2` pool with an unbounded memory limit (i.e., `List[]` in line 19). Now, the three-element allocation succeeds (line 20)[1]. Then, we mutate the tail of the first element to point to a newly allocated element in the same pool, which also succeeds (line 21). In the moment of the mutation, the old subtree (containing values "20" and "30") is completely removed from memory. The print command (line 22) outputs "50", displaying the `head` of the new second element. Again, the end of the block (line 24) deallocates the pool along with all of its elements.

In CÉU, recursive data types have a number of restrictions. Given that mutations deallocate whole subtrees, data types cannot represent general graphs with cycles, but only tree-like structures. Also, elements in different pools cannot be mixed. Finally, pointers to subtrees (i.e., weak references) must be observed via the `watching` construct, as they can be invalidated at any time (to be discussed in Section 2.2).

```
1   data List with
2      tag NIL ();
3   or
4      tag CONS (int head, List tail);
5   end
6
7   do
8      pool List[1] lst1;
9      lst1 = new List.CONS(10,
10                    List.CONS(20,
11                     List.CONS(30,
12                      List.NIL()));
13     _printf("%d, %d\n", lst1.CONS.head,
14                         lst1.CONS.tail.NIL);
15        // prints 10, 1
16  end
17
18  do
19     pool List[] lst2;
20     lst2 = new CONS(10, CONS(20, CONS(30, NIL())));
21     lst2.CONS.tail = new CONS(50, NIL());
22     _printf("%d\n", lst2.CONS.tail.CONS.head);
23        // prints 50 (20 and 30 have been freed)
24  end
```

Figure 2: A recursive `List` data type definition (lines 1–5) with uses (lines 7–16 and 18–27).

```
1   pool List[3] lst = <...>;  // [10, 20, 30]
2
3   var int sum =
4      traverse e in lst do
5         if e:NIL then
6            escape 0;
7         else
8            var int sum_tail = traverse e:CONS.tail;
9            escape sum_tail + e:CONS.head;
10        end
11     end;
12
13  _printf("sum = %d\n", sum);
14        // prints 60
```

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

2.2 Traversing Data Types

CÉU introduces a structured mechanism to traverse recursive data types. The `traverse` construct integrates with the synchronous execution model of CÉU and supports nested control compositions, such as `await` and all `par` variations.

The code in Figure 3 creates a list (line 1) and traverses it to calculate the sum of elements (lines 3–11). The `traverse` block (lines 4–11) starts with the element `e` pointing to the root of the list `lst`. The `escape` statement (lines 6 and 9) returns a value to the enclosing assignment to `sum` (line 3). A `NIL` list[2] has `sum=0` (lines 5–6). A `CONS` list needs to calculate the `sum` of its tail recursively, invoking `traverse` again, which will create a nested instance of the enclosing `traverse` block (lines 4–11), now with `e` pointing to `e:CONS.tail` (line 8). Only after its complete subtree is traversed recursively that it adds its `head` and returns (line 9).

When used without event control mechanisms, as in this simple example, a `traverse` block is equivalent to an anonymous closure called recursively. However, `traverse` complies with the event system and memory management discipline

[1]To save space, in the next examples we omit the data type prefix in tags (e.g., `List.CONS` becomes `CONS`).

[2]The operator '`:`', as in `e:NIL`, is equivalent to C's '`->`'.

```
1   pool List[3] lst = <...>;  // [10, 20, 30]
2
3   var int sum =
4      traverse e in lst do
5         if e:NIL then
6            escape 0;
7         else
8            watching e do
9               _printf("me = %d\n", e:CONS.head);
10              await 1s;
11              var int sum_tail = traverse e:CONS.tail;
12              escape sum_tail + e:CONS.head;
13           end
14           escape 0;
15        end
16     end;
17
18  _printf("sum = %d\n", sum);
19        // prints 60
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34  .
```

CODE-1: Original code (with `traverse`)

```
1   pool List[3] lst = <...>;  // [10, 20, 30]
2
3   class Frame with
4      pool Frame[3]& frames;
5      var  Frame&    parent;
6      pool List[3]*  e;
7   do
8      watching this.parent do
9         if e:NIL then
10           escape 0;
11        else
12           watching e do
13              _printf("me = %d\n", e:CONS.head);
14              await 1s;
15              var Frame* frame = spawn Frame(this.frames,
16                                             this,
17                                             e:CONS.tail);
18                                 in this.frames;
19              var int sum_tail = await *frame;
20              escape sum_tail + e:CONS.head;
21           end
22           escape 0;
23        end
24     end
25     escape 0;
26  end
27
28  pool Frame[3] frames;
29  var Frame* frame = spawn Frame(frames, this, lst)
30                     in frames;
31  var int sum = await *frame;
32
33  _printf("sum = %d\n", sum);
34        // prints 60
```

CODE-2: Expanded code (without `traverse`)

Figure 4: Calculating the *sum* of a list, one element each second. The `traverse` construct is a syntactic sugar that can be "desugared" with explicit organisms.

of CÉU and is an abstraction defined in terms of a more fundamental concept [9]: *organisms*, which are objects with concurrent trails of execution (akin to Simula [1]); and orthogonal abortion, which handles cancellation of trails maintaining memory consistency. Figure 4 depicts the expansion of the `traverse` construct.

The example in *CODE-1* of Figure 4 extends the body of the previous example in Figure 3 with reactive behavior. Now, for each recursive iteration, we print the current `head` (line 9) and `await` 1 second (line 10) before traversing the `tail` (line 11). Note that the last iteration of `traverse` is waiting for one second before printing "30", with all previous iterations blocked and retaining their full state of execution. Furthermore, this code can be a subprogram of a larger program with other trails in parallel, all of which remain reactive during the incremental traversal.

CÉU enforces at compile time that all accesses to a pointer that cross `await` statements are protected with an enclosing `watching` block [9]. The `watching` aborts its nested block if the referred object is released from memory. This ensures that if concurrent side effects affect the pointed object, no code uses the stale pointer, because the whole block is aborted. With the protection of the `watching` block (lines 8–13), if the referred element `e` (line 8) is released from memory due to a mutation in the list during the awaiting period (line 10), we simply ignore the whole subtree and return 0 (line 14).

CODE-2 is the equivalent expansion of *CODE-1* without the `traverse` construct. Because it contains concurrency constructs (i.e., `await` and `watching`), the body of the `traverse` (*CODE-1*: 5–15) must be abstracted in an organism of the `Frame` class (*CODE-2*: 3–26), which is analogous to a "stack frame" of standard programming languages with subroutines. Likewise, the `pool` of frames (*CODE-2*: 28) is analogous to a runtime "call stack". We limit the number of stack frames to match the maximum number of elements to traverse (*CODE-1*: 1 and *CODE-2*: 1,28). Therefore, to "call" the first `traverse` iteration, we dynamically `spawn` a `Frame` instance into the `frames` pool (*CODE-2*: 29–30). Then, we immediately `await` the termination of this `frame`, which is in the lowest level of the "call stack", i.e., only after the `traverse` rolls back and clears the "call stack" that we acquire the `sum` and print it (*CODE-2*: 31–34).

A `Frame` receives three arguments in the constructor (*CODE-2*: 4–6): a reference to a `pool` (to recursively `spawn` new frames); a reference to its parent frame (to handle abortion); and a pointer to the subtree of the data type (to be able to manipulate it). The `Frame` constructor for the first "call" (*CODE-2*: 29–30) receives the static pool of frames, the running organism as the parent (i.e., `this`), and the tree of the data type passed to the original `traverse` (*CODE-1*: 4). The `Frame` constructor for recursive "calls" (*CODE-2*: 15–18) receives the same pool of frames, the current "stack frame" as parent, and the subtree of the data type passed to the original recursive `traverse` invocation (*CODE-1*: 11).

The `Frame` body (*CODE-1*: 8–25) always aborts with the `parent` frame termination. Given that a `traverse` body can execute (and terminate) trails running concurrently with the recursive invocation, the enclosing `watching` guarantees that the hierarchy in the "call stack" will be preserved (i.e., that there are no orphan frames executing). The remaining code is almost the same in the original `traverse` body and in the `Frame` body (*CODE-1*: 5–15 and *CODE-2*: 9–23). The only exception is the recursive invocation of the `traverse` (*CODE-1*: 11), which expands to a `spawn` of a `Frame` (*CODE-2*: 15–18) with similar behavior to the first invocation (*CODE-2*: 29–30).

As the expansion illustrates, three aspects make `traverse` fundamentally different from an recursive function calls:

1. Each `traverse` invocation spawns a new organism which can execute concurrently with other parts of the application. Also, each organism itself can invoke multiple concurrent trails, adding more complexity in comparison to standard functions [9].

2. Traversal is declared in terms of a specific lexically-scoped memory pool for a data structure. Therefore, we can infer at compile time the maximum traversal depth if the data is bounded (e.g., `List[3] lst`). Enforcing execution limits is an important requirement for constrained and real-time embedded systems, which is the original application domain of CÉU [8]. In addition, given that the pool of frames is expanded in the same lexical scope, when the data goes out of scope, all stack frames are automatically aborted [9].

3. The execution body of a `traverse` block is implicitly wrapped by a concurrency construct that watches for mutations of the current node. In practice, this means that it reacts consistently if another trail of execution modifies the data structure being traversed.

We believe that the `traverse` construct, more than a simple convenience, considerably reduces the complexity of programs, handling a complex hierarchy of behaviors associated with recursive data types automatically.

3. APPLICATIONS

In this section, we present three applications that explore the reactive and incremental nature of the `traverse` construct. We start with standard *Behavior Trees* used in video games for AI modeling [4, 2]. Then, we show a *Logo Turtle* [7] that can execute commands in parallel (e.g., `move` and `rotate`). Finally, we extend the Turtle example with a dynamic and concurrent queue of commands that can affect the running program.

3.1 Behavior Trees

Behavior Trees are a family of DSLs used for game AI. The term is loose, because different games use different languages. For our purposes it indicates an interpreted domain-specific language for concurrent creature behavior that includes sequence and selection combinators.

The `SEQ` can be understood as short-circuit evaluation of an 'and', while the `SEL` corresponds to an 'or'. This skeleton is extensible with leaves to test properties, set properties, perform animations and sounds, etc., and is an effective alternative to finite state machines for authoring game AI. However, because the evaluation of each tree extends across many frames, writing behavior tree nodes and leaves in other languages is an exercise in stack-ripped event-driven

```
1  data BTree with
2     tag NIL ();
3  or
4     tag SEQ (BTree first, BTree second);
5  or
6     tag SEL (BTree first, BTree second);
7  or
8     tag LEAF (Leaf& leaf);
9  end
10
11 class BTreeInterpreter with
12    pool BTree[]& btree;
13 do
14    var int ret =
15       traverse t in btree do
16          if t:SEQ then
17              var int ok = traverse t:SEQ.first;
18              if ok == 0 then
19                  escape ok;
20              end
21              ok = traverse t:SEQ.second;
22              escape ok;
23          else/if t:SEL then
24              var int ok = traverse t:SEL.first;
25              if ok != 0 then
26                  escape ok;
27              end
28              ok = traverse t:SEL.second;
29              escape ok;
30          else/if t:LEAF then
31             var int ret =
32                 do LeafHandler(t:LEAF.leaf);
33             escape ret;
34          end
35       end;
36    escape ret;
37 end
```

Figure 5: A simple generic grammar of behavior trees with *sequence* and *selector* composite nodes with a straightforward interpreter.

programming [5]. By lowering the barrier to writing custom nodes and leaves, CÉU lightweight event control mechanisms make behavior trees more usable.

Figure 5 describes a generic grammar for behavior trees (lines 1–9). The `SEQ` and `SEL` tags (lines 4 and 6) are recursive and behave as described above. The `LEAF` tag (line 8) receives a reference to a `Leaf` data type, which is defined externally, carrying opaque values that are specific to the application domain. The interpreter for tree instances is abstracted in a class definition (lines 11–37), which receives the tree to execute as an argument (line 12), acquires the return status of the traversal (line 14) and returns it as its final result (lines 36). The traversal takes into account the semantics of the composite nodes. For the `SEQ` tag (lines 16–22), we traverse the `first` subtree (line 17) and only if it succeeds, we traverse the `second` subtree (line 21). For the `SEL` tag (lines 23–29), we traverse the `first` subtree (line 24) and only if it fails, we traverse the `second` subtree (line 28). Finally, the `LEAF` tag does the real work and is domain specific. The `do Class` syntax (line 31–32) creates an anonymous and lexically scoped organism of the referred class and awaits its termination. The organism itself can contain any valid code in CÉU and execute for an arbitrary amount of time [9]. The tree traversal hangs and waits for its return status (line 33). This is the expected behavior given that `SEQ` and `SEL` nodes want to execute their children in sequence.

The blocks world is a classical planning domain [?]. The leaves of a behavior tree in an extended blocks world domain

```
1  pool BTree[] btree = new SEQ(SEL(SEQ(
2        LEAF(SENSE_ON_TABLE(C)),
3        LEAF(MOVE_BLOCK_TO_BLOCK(B, C, A))),
4      SEQ(
5        LEAF(MOVE_TO_TABLE(C, B)),
6        LEAF(MOVE_FROM_TABLE(B, A)))),
7    LEAF(MOVE_FROM_TABLE(C, B)));
```

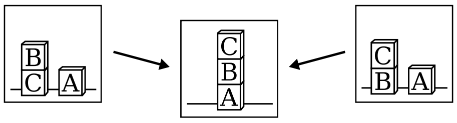

Figure 6: A blocks world behavior tree.

might include sensor leaves that either succeed or fail, as well as actuator leaves that move blocks and unconditionally succeed.

The tree in Figure 6 is based on output from Contingent-FF [3], and the blocks domain extended with sensor actions is one of Contingent-FF's benchmark problems. Here, we want to achieve an ABC stack. We assume we pinned down the situation to two possibilities. We use a `SEL` node (line 1) to go through them, and a sensor leaf (line 2) to decide which strategy is appropriate: If it senses that C directly on top of the table, then we move B, then C, then A (line 3). If the sensor does not succeed, we move to the second sequence of the selection (lines 4-6): we move C from B to the table and B from the table to A. Finally, completing the outer sequence that applies to both cases, we move C from the table to B (line 7), thus achieving an ABC stack.

This illustrates how the behavior tree can exhibit goal-directed behavior. The goal-directedness is not automatic; rather, behavior trees are authored by designers to exhibit goal-directedness. The `traverse` feature allows behavior authored at runtime, which is one of the crucial features of behavior trees. Alternatively, `traverse` allows CÉU to to consume at runtime the output of CPU-intensive algorithms such as planners.

3.2 Logo Turtle

Our second example is an interpreter for a simple variant of the classic Logo turtle-graphics interpreter [7]. The aim of this example is to demonstrate parallel traversal. In our variant, we can instruct the turtle to move and rotate in parallel, tracing curves.

```
1  data Command with
2     tag NOTHING ();
3  or
4     tag SEQ (Command first, Command second);
5  or
6     tag REPEAT (int times, Command command);
7  or
8     tag MOVE (int pixels);
9  or
10    tag ROTATE (int angle);
11 or
12    tag AWAIT (int ms);
13 or
14    tag PAROR (Command first, Command second);
15 end
16
17 class CommandInterpreter with
18    var  Turtle&    turtle;
19    pool Command[]* cmds;
20 do
21    traverse cmd in cmds do
22       watching cmd do
23          if cmd:SEQ then
24             traverse cmd:SEQ.first;
25             traverse cmd:SEQ.second;
26
27          else/if cmd:REPEAT then
28             loop i in cmd:REPEAT.times do
29                traverse cmd:REPEAT.command;
30             end
31
32          else/if cmd:MOVE then
33             do TurtleMove(turtle,
34                           cmd:MOVE.pixels);
35
36          else/if cmd:ROTATE then
37             do TurtleRotate(turtle,
38                             cmd:ROTATE.angle);
39
40          else/if cmd:AWAIT then
41             await (cmd:AWAIT.ms) ms;
42
43          else/if cmd:PAROR then
44             par/or do
45                traverse cmd:PAROR.first;
46             with
47                traverse cmd:PAROR.second;
48             end
49          end
50       end
51    end
52 end
```

Figure 7: DSL grammar and interpreter for a Logo turtle.

```
1  pool Command[] cmds =
2     new PAROR(
3           AWAIT(1000),
4           PAROR(MOVE(300), ROTATE(180)));
5
6  var Turtle turtle;
7  do CommandInterpreter(turtle, cmds);
```

Figure 8: A turtle program.

Figure 7 presents the data type `Command` (lines 1–9), which specifies the abstract syntax of our Logo variant. As in traditional Logo, commands can execute in sequence through the `SEQ` tag (line 4), and can also repeat a number of times through the `REPEAT` tag (line 6). Our variant extends the `MOVE` and `ROTATE` commands to take as arguments the speed at which they should affect the turtle (lines 8 and 12). For example, a `Command.MOVE(300)` node directs the turtle to move at the speed of 300 pixels per second, indefinitely. Therefore, the only way to make the turtle stop moving or rotating is through two CÉU-like extensions added to our Logo variant: The `AWAIT` tag (line 12) simply awaits a given number of milliseconds. The `PAROR` tag (line 14), modeled after the CÉU construct `par/or`, launches two commands in parallel, and aborts both of them as soon as one of them finishes. As an example, the program in Figure 8 makes the turtle to move along a semicircle.

The interpreter for the commands is also abstracted in a class definition (lines 17–52 of Figure 7). It holds as attributes a reference to a `Turtle` object (which implements the UI) and reference to the commands (lines 18–19). The execution body of the class uses the `traverse` construct to interpret the commands (lines 21–51). The `SEQ` tag (lines

```
data Queue with
  tag NIL ();
or
  tag ROOT (Queue running,
            Queue waiting,
            Queue tmp);
or
  tag ITEM (Command cmds,
            Queue prv);
end
.
```

CODE-3: `Queue` type

```
 1 traverse qu in queue do
 2   watching qu do
 3     if qu:ROOT then
 4       loop do
 5         par/and do
 6           traverse qu:ROOT.running;
 7         with
 8           await qu:ROOT.waiting;
 9         end
10         qu:ROOT.running =
11           qu:ROOT.waiting;
12         qu:ROOT.waiting =
13           new ITEM(NOTHING(), NIL());
14       end
15     else/if qu:ITEM then
16       traverse qu:ITEM.prv;
17       do CommandInterpreter(
18             turtle, qu:ITEM.cmds);
19     end
20   end
21 end
```

CODE-4: Queue traversal

```
 1 input (char*,int,int) ENQUEUE;
 2 every (cmd,vel,time) in ENQUEUE do
 3   if _strcmp(cmd,"move")==0 then
 4     queue.ROOT.tmp =
 5       new ITEM(
 6             PAROR(
 7               MOVE(vel),
 8               AWAIT(time)),
 9             NIL());
10   else/if _strcmp(cmd,"rotate")==0 then
11     <...> // analogous to the MOVE above
12   end
13   queue.ROOT.tmp.ITEM.prv =
14     queue.ROOT.waiting.ITEM.prv;
15   queue.ROOT.waiting = queue.ROOT.tmp;
16 end
17
18
19
20
21 .
```

CODE-5: Enqueuing commands

Figure 9: Queue extension for the Turtle DSL of Figure 7.

23–25) traverses each of its child commands in sequence (in contrast with the `BTreeInterpreter`, it does not handle failures). The `REPEAT` tag (lines 27–30) traverses its command the specified number of times. The `MOVE` and `ROTATE` tags (lines 32–34 and 36–38) relies on predefined classes of organisms to update the position and orientation of the `turtle` received as argument in the constructor (line 18). The `AWAIT` tag (lines 40–41) simply causes the current trail of execution of the interpreter to await the given amount of time. Finally, the `PAROR` tag (lines 40–48) uses the `par/or` construct to traverse both subcommands at the same time. As per the semantics of `par/or`, as soon as one of the subtrees terminate its execution, the other one is aborted.

Note that the entire interpreter block is surrounded by a `watching` construct (line 22). As discussed in Section 2.2, the Céu compiler enforces the presence of a guard, due to the use of the `cmd` pointer in code that spans multiple reactions. This ensures clean abortion in case the AST is mutated by code running other trails.

3.3 Enqueuing Commands

All examples so far create a fixed tree that does not vary during traversal. Figure 9 extends the Turtle application with a queue of pending commands to execute after the running commands terminate.

We define a new `Queue` data type in (*CODE-3*): `ROOT` has a `running` subtree with the running commands, a `waiting` queue of pending commands to execute, and a `tmp` node that allows in-place manipulation of the tree. Given that all newly allocated nodes must reside in a pool, the `tmp` node represents a pointer TODO. `ITEM` represents a queue item and contains a `cmd` subtree with the command to execute, and a `prv` queue item pointing to an older item that should execute first (i.e., the queue is in reverse order). We define a new `Queue` data type in *CODE-3*: The tag `ROOT` has a `runnning` subtree with the running commands, a `waiting` queue of pending commands to execute afterwards, and a `tmp` node to allows in-place manipulation of the tree (to be discussed further). The `ITEM` tag represents a queue item and contains a `cmds` subtree with the commands to execute, and a `prv` queue item pointing to an older item that should execute first (i.e., the queue is in reverse order). As Figure 10 illustrates in box `0`, a queue instance should have a single `ROOT` node with linked lists of `ITEM` nodes in the `running` and `waiting` fields. Except on command creation, the `tmp` field is always `NIL`.

The `queue` traversal in *CODE-4* handles the tags `ROOT` (lines 3–16) and `ITEM` (lines 17–20). The `ROOT` traversal is a continuous `loop` that executes the `running` subtree and swaps it on termination with the `waiting` queue. The `par/and` (lines 5–9) ensures that that the swap only occurs after the current `running` commands terminate (line 6) *and* something (in parallel) mutates the `waiting` subtree (line 8), meaning that the queue is no longer empty. The swapping process (lines 10–15) is illustrated in Figure 10 in the respective boxes (0–2):

0. The initial state assumes pre-existing `running` and `waiting` items.

1. Lines 10–11 assign the `waiting` subtree to the `running` field (mark `(a)`), releasing the old subtree (mark `(b)`)). The `waiting` field is automatically set to `NIL` (mark `(c)`).

2. Lines 12–15 assign a new neutral `ITEM` (with a dummy `NOTHING` command in the `cmds` field) to the `waiting` queue (mark `(d)`).

After the swapping process, the `loop` restarts and traverses the new `running` commands. The `ITEM` traversal is straightforward: first we traverse the previous item (line 18), and then we reuse the `CommandInterpreter` class of Figure 7 to traverse the commands (line 19–20).

Even though this example mutates the `running` field only *after* its traversal terminates, it is safe to do an arbitrary mutation at any time. Note that the compiler enforces the use of the `watching` construct (lines 3–22) to enclose the running turtle interpreter (lines 19–20). Hence, if its enclosing `ITEM` (line 17) is mutated, the `watching` will awake and abort the interpreter inside its lexical scope.

The enqueuing of new commands is depicted in *CODE-5*. The external input event `ENQUEUE` (line 1) accepts *move* and *rotate* commands with an associated velocity and time (i.e.,

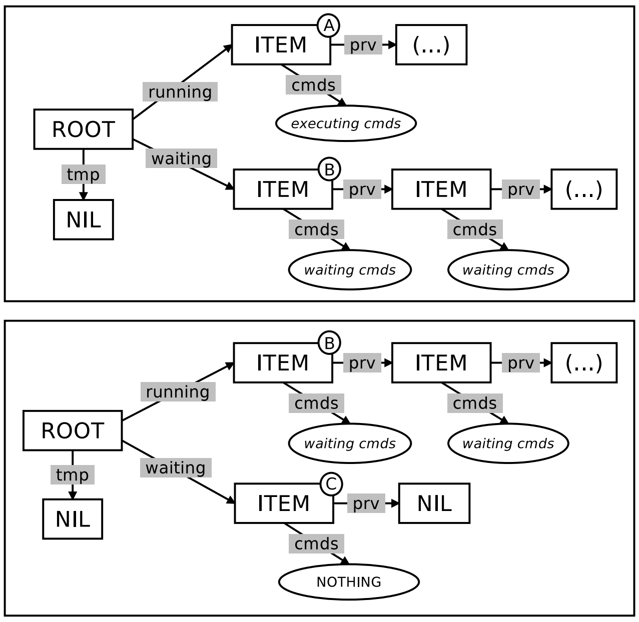

Figure 10: Swapping `waiting` and `running` commands.

`char*,int,int` arguments). The `every` loop reacts to each occurrence of `ENQUEUE`, creating and enqueuing the requested command, as illustrated in Figure 11 (1–3):

0. The initial state assumes the pre-existing neutral `ITEM` in the root of the `waiting` field.
1. Line 4–9 create the new `ITEM`, with the set of commands to `MOVE` the turtle, and assigns it to the `tmp` field (mark `a`).
2. Lines 13–14 move the already `waiting` commands to tail of the `tmp` node (mark `(b)`). Note that the neutral `ITEM` is skipped to avoid the `waiting` root to become `NIL` and awake the `ROOT` node (line 8 of *CODE-4*) before we finish the enqueuing operation. The old location for the moved commands is automatically set to `NIL` (mark `(c)`).
3. Line 15 moves the `tmp` subtree back to the `waiting` field (mark `(d)`), releasing the neutral `ITEM` (mark `(e)`), and notifying the `ROOT` node that the queue is no longer empty. The `tmp` field is automatically set to `NIL` (mark `(f)`).

- TODO: modularization of data type and travesal

4. RELATED WORK

Synchronous languages - data types in synchronous languages

Data structures under constraints: functional/immutable data structures, persistent data structures.

Traversing data structures: syntactic support and expressivity

- Python generators
- Lua iterators (coroutines + generic for)
- generators in Icon (1981 paper "Generators in Icon": keywords include "goal-directed programming, generators, nondeterministic programming, backtracking" – talks about 'control backtracking', which looks conceptually like behavior trees) http://drhanson.s3.amazonaws.com/storage/documents/gen in-icon.pdf

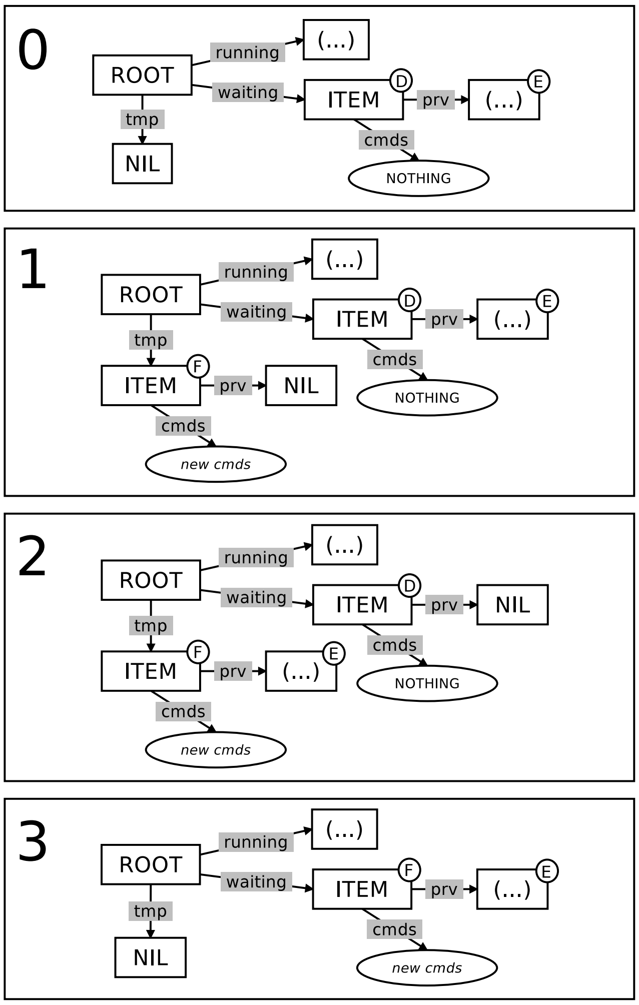

Figure 11: Enqueuing new commands.

5. CONCLUSION

We presented a new construct for traversing recursive data types incrementally, in the context of CÉU, an imperative reactive language with synchronous concurrency. The `traverse` construct encapsulates an idiom for performing recursive traversal by handling each step as a separate trail of execution. This allows parallel traversal using the language's concurrency features, while maintaining its safety properties.

This kind of traversal can be performed in CÉU through the use of organisms (pooled objects which launch their own execution trails) and orthogonal abortion via the `watching` construct. Combining these features to traverse a recursive data structure correctly, however, is not straightforward. Recursing in a way such that parallel constructs can be composed requires each step of the recursion to be a new execution trail. Ensuring that the traversal will not execute on a stale subtree in case the structure is modified requires the nodes to be watched in order to perform abortions. Additionally, by presenting a control construct that is tied to a data structure, we can ensure bounded execution time, in line with the CÉU philosophy. By dealing with these concerns internally in the `traverse` statement, we make reactive traversal as easy to perform correctly as a recursive function call.

In the current implementation of recursive data types in CÉU, we impose restrictions to the kinds of structures that can be represented. The requirement of a tree hierarchy of ownership and move semantics for assignment of structure fields requires care in the design of algorithms manipulating these structures, as illustrated in Section 3.3. This is done to support static memory management with bounded memory pools for allocation and deterministic deallocation. Still, we do not feel that the restrictions are prohibitively limiting. For instance, persistent data structures in functional languages [**?**] operate under tighter design constraints.

Possibilities for future work are the introduction of type arguments for `data`, and investigating possibilities for relaxing CÉU pointer and reference semantics while maintaining their safety properties on recursive data.

6. REFERENCES

[1] O.-J. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[2] C. Hecker. My liner notes for Spore/Spore behavior tree docs. `http://chrishecker.com/My_liner_notes_for_spore`, 2009. [Online; accessed 30-July-2015].

[3] J. Hoffmann and R. Brafman. Contingent planning via heuristic forward search with implicit belief states. In *Proc. ICAPS*, volume 2005, 2005.

[4] D. Isla. Handling complexity in the Halo 2 AI. In *Game Developers Conference, San Francisco*, 2005.

[5] M. N. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX Annual Technical Conference*, pages 87–100, 2007.

[6] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, New York, NY, USA, 1998.

[7] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, Inc., New York, NY, USA, 1980.

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

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