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 domain 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 [12].

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 [14, 15] is an imperative, concurrent and reactive language in which lines of execution, known as *trails*, react together continuously and in synchronous steps to external stimuli. Céu supports mutable data, with static memory and safe pointer manipulation. These features are incompatible with garbage-collected immutable data structures, as well as with general records with arbitrary pointers such as *structs* in C.


```
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.

The solution to this problem is twofold, with data and control aspects. For data management, we introduce a restricted form of recursive data types that supports trees (but not general graphs). To control reactive behavior, we propose a structured mechanism that can traverse data types safely and incrementally, in successive reactions to events. After we present the design of these constructs, we discuss three applications in the domain of control-oriented DSLs.

2. CÉU CONSTRUCTS

The introductory example[1] in Figure 1 gives a general flavor of Céu. It first defines an input event `RESET` (line 1), a shared variable `v` (l. 2), and starts two trails with the `par` construct (l. 3-14): the first (l. 4-8) increments variable `v` on every second and prints its value on screen; the second (l. 10-13) resets `v` on every external request to `RESET`.

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 *atomically*, one after the other, until they await again or terminate. As a consequence, all consecutive operations to shared variable `v` in Figure 1 are atomic because reactions to events `1s` and `RESET` can never interrupt each other. If multiple trails re-

[1]A screencast of all examples in the paper is available at `https://vimeo.com/135297440`.

act 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 [14].

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 (l. 1–5). The first tag of recursive types has a special meaning as the union *null* type. In the example, the tag `NIL` also represents an empty list (l. 2). The second tag, `CONS` (l. 4), receives two arguments in the constructor: an integer value for the `head` field and the rest of the list in the recursive field `tail`.

In the first block of the example (l. 7–16), we declare a `pool` of `List` objects of size 1 (l. 8). All recursive `data` instances must reside in an explicit memory pool, which have static memory management based on its enclosing lexical scope (l. 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` (l. 8). Then, we use the `=new` construct which performs allocation and assignment at the same time (l. 9–12): it attempts to dynamically allocate a list of three elements (10, 20, and 30), and assigns the result to the *l-value*. The destination memory pool for the allocation is inferred from the prefix of the *l-value* (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 (l. 13–14) outputs "`10, 1`": the `head` of the first element, and *true* for the `NIL` check of the second element. Finally, the end of the block (l. 16) deallocates the pool along with all elements inside it.

In the second block (l. 18–24), we declare the `lst2` pool with an unbounded memory limit (i.e., `List[]` in line 19). Now, all allocations succeed (l. 20)[2]. Then, we mutate the tail of the first element to point to a newly allocated element in the same pool, which also succeeds (l. 21). At the moment of the mutation, the old subtree (containing values "`20`" and "`30`") is completely removed from memory. The print command (l. 22) outputs "`50`", displaying the `head` of the new second element. Again, the end of the block (l. 24) deallocates the pool along with all of its remaining 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).

2.2 Traversing Data Types

Céu introduces the `traverse` structured mechanism for traversing recursive data types incrementally. The code in Figure 3 creates a list (l. 1) and traverses it to calculate the sum of elements (l. 3–11). The `traverse` block (l. 4–11)

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

```
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 (l. 1–5) with uses (l. 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); // prints 60
```

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

starts with the element `e` pointing to the root of the list `lst`. The `escape` statement (l. 6 and 9) returns a value to the enclosing assignment to `sum` (l. 3). A `NIL` list[3] has `sum=0` (l. 5–6). A `CONS` list needs to calculate the `sum` of its tail recursively, invoking `traverse` again (l. 8), which will create a nested instance of the enclosing `traverse` block (l. 4–11), now with `e` pointing to `e:CONS.tail`. Only after the complete recursive traversal of its subtree that the `CONS` clause adds its `head` and returns (l. 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 of Céu and is an abstraction defined in terms of a more fundamental concept, *organisms* [15], which are objects with concurrent trails of execution (akin to Simula [3]). Figure 4 depicts the expansion of the `traverse` construct.

The example in *CODE-1* (ⓒ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` (l. 9) and `await` 1 second (l. 10) before traversing the `tail` (l. 11). Note that while nested iterations of `traverse` await 1 second, all previous iterations are blocked, retaining their full state of execution. Furthermore, the example could be part of a larger program with other trails

[3]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
35
36
37
38
39 .
```

ⓒ1 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  Scope*    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         do
16           var Scope scope;
17           var Frame* frame = spawn Frame(this.frames,
18                                          &scope,
19                                          e:CONS.tail);
20                              in this.frames;
21           var int sum_tail = await *frame;
22         end
23         escape sum_tail + e:CONS.head;
24       end
25       escape 0;
26     end
27   end
28   escape 0;
29 end
30
31 do
32   pool Frame[3] frames;
33   var Scope scope;
34   var Frame* frame = spawn Frame(frames, &scope, lst)
35                      in frames;
36   var int sum = await *frame;
37 end
38
39 _printf("sum = %d\n", sum); // prints 60
```

ⓒ2 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.

in parallel, all of which would 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. The `watching` automatically aborts its nested block when the referred object is released from memory [15]. 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 (l. 8–13), if the element `e` (l. 8) is released from memory due to a mutation in the list during the awaiting period (l. 10), we simply ignore the whole subtree and return 0 (l. 14).

CODE-2 (ⓒ2) is the equivalent expansion of ⓒ1 without the `traverse` construct. Because it contains concurrency constructs (i.e., `await` and `watching`), the body of the `traverse` (ⓒ1: 5–15) is abstracted in an organism of the `Frame` class (ⓒ2: 3–29), which is analogous "stack frames" of subroutines in standard programming languages. Likewise, the `pool` of frames (ⓒ2: 32) is analogous to a runtime "call stack". We limit the number of stack frames to match the maximum number of elements to traverse (ⓒ1: 1 and ⓒ2: 1,32). Therefore, to "call" the first `traverse` iteration, we dynamically `spawn` a `Frame` instance into the `frames` pool (ⓒ2: 34–35). Then, we immediately `await` the termination of this `frame` (ⓒ2: 36). Only after the `traverse` rolls back and clears the whole call stack that we acquire the `sum` and print it (ⓒ2: 36–39).

A `Frame` receives three arguments in the constructor (ⓒ2: 4–6): a reference to a `pool` (to recursively `spawn` new frames); a pointer to its parent scope (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 (ⓒ2: 34–35) receives the static pool of frames, a dummy `scope` organism attached to the current scope, and the original tree to traverse (ⓒ1: 4). The `Frame` constructor for recursive calls (ⓒ2: 17–20) receives the same pool of frames, another dummy `scope` as parent, and the original subtree in the recursive invocation (ⓒ1: 11). The `Frame` body (ⓒ1: 8–28) always aborts with the `parent` scope termination. Given that a `traverse` body can possibly execute (and terminate) trails running concurrently with the recursive invocation, the enclosing `watching` guarantees that the hierarchy in the call stack is 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 (ⓒ1: 5–15 and ⓒ2: 9–26), with the exception of the recursive invocation explained above (ⓒ2: 17–20).

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

1. Each `traverse` invocation spawns a new organism for the frame which can execute concurrently with other parts of the application. Also, each frame itself can

contain multiple concurrent trails (to be illustrated in Section 3), adding more complexity in comparison to standard functions [15].

2. A `traverse` is attached to a lexically-scoped memory pool for specific 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 bounded limits is an important requirement for constrained and real-time embedded systems, which is the original application domain of Céu [14]. In addition, given that the pool of frames is expanded in the same lexical scope, when the associated data goes out of scope, all stack frames are automatically aborted [15].
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 automatically hierarchy of behaviors associated with recursive data types.

3. APPLICATIONS

In this section, we present three applications that explore the reactive nature of the `traverse` construct. We start with *Behavior Trees* used in video games for AI modeling. Then, we show a *Logo Turtle* 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 [9, 6]. The DSLs vary between languages, but they usually include sequence (`SEQ`) and selection (`SEL`) combinators to model concurrent creature behavior. 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 trees extend across multiple game frames, specifying node behaviors in generic languages via event-driven programming becomes a challenge due to "stack ripping" [10]. 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 (l. 1–9). The `SEQ` and `SEL` tags (l. 4 and 6) are recursive and behave as described above. The `LEAF` tag (l. 8) receives a reference to an opaque `Leaf` data type, which is defined externally and is specific to the application domain. The interpreter for trees is abstracted in a class definition (l. 11–37) and receives the tree to traverse as the single argument (l. 12). The body acquires the return status of the traversal (l. 14) and returns it as the final result (l. 36). For the `SEQ` tag (l. 16–22), we traverse the `first` subtree (l. 17) and only if it succeeds, we traverse the `second` subtree (l. 21). For the `SEL` tag (l. 23–29), we traverse the `first` subtree (l. 24) and only if it fails, we traverse the `second` subtree (l. 28). Finally, the `LEAF` tag (l. 30–33) delegates the behavior to another class, which does real work and is domain specific. The `do`

```
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 grammar of behavior trees with `SEQ` and `SEL` nodes and a straightforward interpreter.

`Class` syntax (l. 31–32) creates an anonymous and lexically scoped organism and awaits its termination to return the final status (l. 33). The organism itself can contain any valid code in Céu (including parallel compositions) and executes for an arbitrary amount of time [15].

As an example of a domain, the *blocks world* is a classical planning domain in AI [17]. The tree in Figure 6 is based on the output from a Contingent-FF benchmark that extends the blocks domain with sensor actions [7]. We want to achieve an `ABC` stack and assume two possibilities, as illustrated in the figure. We use a `SEL` node (l. 3) with a sensor leaf (l. 4) to decide which strategy is appropriate: If `C` is not sensed on top of the table, we first move it to the table (l. 5). Then, in both situations, we stack `B` on top of `A`, and `C` on top of `B` (l. 6–7). The example illustrates how the behavior tree can exhibit goal-directed behavior specified directly by domain designers.

3.2 Logo Turtle

Our second example is an interpreter for a simple variant of the classic Logo turtle-graphics interpreter [13]. 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.

Figure 7 presents the data type `Command` (l. 1–15), which specifies the abstract syntax of our Logo variant. As in traditional Logo, commands can execute in sequence through the `SEQ` tag (l. 4), and can also repeat a number of times through the `REPEAT` tag (l. 6). Our variant extends the `MOVE` and `ROTATE` commands to take as arguments the speed at

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

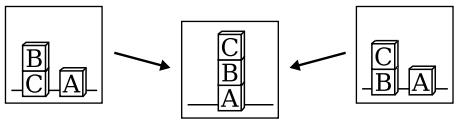

Figure 6: A *blocks world* behavior tree.

which they should affect the turtle (l. 8,10). 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 (l. 12) simply awaits a given number of milliseconds. The `PAROR` tag (l. 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 lines 54–60 makes the turtle to move along a semicircle.

The interpreter for the commands is also abstracted in a class definition (l. 17–52). It holds as attributes a reference to a `Turtle` object (which implements the UI) and a reference to the commands (l. 18–19). The execution body of the class uses the `traverse` construct to interpret the commands (l. 21–51). The `SEQ` tag (l. 23–25) traverses each of its subcommands in sequence (in contrast with `BTreeInterpreter`, it does not handle failures). The `REPEAT` tag (l. 27–30) traverses its subcommand the specified number of times. The `MOVE` and `ROTATE` tags (l. 32–38) rely on predefined classes of organisms to update the position and orientation of the `turtle` (received in line 18). The `AWAIT` tag (l. 40–41) simply causes the current iteration of the `traverse` to await the given amount of time. Finally, the `PAROR` tag (l. 43–48) 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 (l. 22). As discussed in Section 2.2, the CÉU compiler enforces the presence of the 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 in other trails.

3.3 Enqueuing Commands

All examples so far create a fixed tree that does not vary during traversal. Figure 8 extends the Turtle application with a queue of pending commands to execute.

We define a new `Queue` data type in *CODE-3*: The `ROOT` tag (l. 4–6) has a `running` subtree of commands, a `waiting` queue of pending commands, and a `tmp` node to allow in-place manipulation of the tree (to be discussed further). The `ITEM` tag (l. 8–9) represents a queue item and contains a `cmds` subtree with the actual commands to execute (as described in Figure 7: 1–15), and a `prv` queue item pointing to an older item (i.e., the queue is in reverse order). As Figure 9 illustrates in box `0`, a queue instance has a single `ROOT` node with linked lists of `ITEM` nodes in the `running` and `waiting`

```
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
53
54 pool Command[] cmds =
55    new PAROR(
56          AWAIT(1000),
57          PAROR(MOVE(300), ROTATE(180)));
58
59 var Turtle turtle;
60 do CommandInterpreter(turtle, cmds);
```

Figure 7: Grammar, interpreter, and sample program for a Logo turtle DSL.

fields. Except when creating a new command, the `tmp` field is always `NIL`.

We define the `queue` traversal in *CODE-4*. The `ROOT` traversal (l. 3–14) is a continuous `loop` that executes the `running` subtree and swaps it with the `waiting` queue on termination. The `par/and` (l. 5–9) ensures that that the swap only occurs after the current `running` commands terminate (l. 6) *and* something (in parallel) mutates the `waiting` subtree (l. 8), meaning that the queue is no longer empty. The swapping process (l. 10–13) is illustrated in Figure 9:

- The initial state (box `0`) assumes pre-existing `running` and `waiting` items.
- Lines 10–11 assign the `waiting` subtree (marked Ⓑ)

```
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              NOTHING(),
7              ITEM(
8                PAROR(
9                  MOVE(vel),
10                 AWAIT(time)),
11               NIL()));
12   else/if _strcmp(cmd,"rotate")==0 then
13     <...> // analogous to the MOVE above
14   end
15   queue.ROOT.tmp.ITEM.prv.ITEM.prv =
16     queue.ROOT.waiting.ITEM.prv;
17   queue.ROOT.waiting = queue.ROOT.tmp;
18 end
19
20
21 .
```

CODE-5: Enqueuing commands

Figure 8: Queue extension for the Turtle DSL of Figure 7.

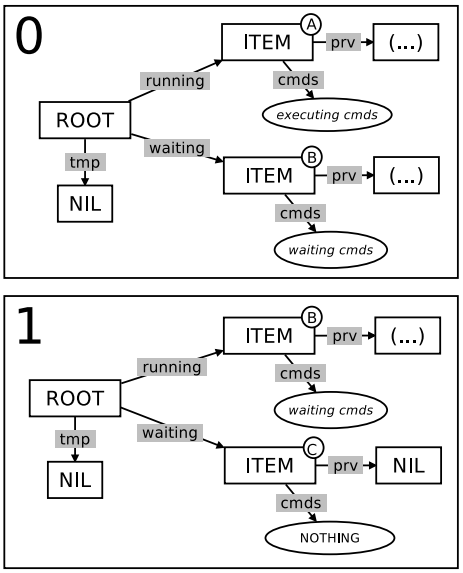

Figure 9: Swapping `waiting` and `running` commands.

to the `running` field, releasing the old subtree (Ⓐ)). Recursive data types Céu have *move semantics*, hence, the `waiting` field is automatically set to the union null type (i.e., `NIL`).

- Lines 12–13 assign a new neutral `ITEM` (box `1`: Ⓒ, with a dummy command `NOTHING`) to the `waiting` field, completing the swapping operation.

After the swapping process, the `loop` restarts and traverses the new `running` commands (l. 4). The `ITEM` traversal (l. 15–18) is straightforward: first we traverse the previous item (l. 16), and then we reuse the `CommandInterpreter` class of Figure 7 to traverse the commands (l. 17–18).

Even though this example mutates the `running` field only *after* its traversal terminates (l. 10–11), it is safe to do an arbitrary mutation at any time. Note that the compiler enforces the use of the `watching` construct (l. 2) which encloses the running turtle interpreter (l. 17–18). Hence, if its enclosing `ITEM` (l. 15) is mutated, the `watching` will awake and abort the interpreter running inside the lexical scope.

The enqueuing of new commands is depicted in *CODE-5*. The external input event `ENQUEUE` (l. 1) accepts *"move"* and *"rotate"* strings with an associated velocity and time (i.e., "`char*,int,int`" arguments). The `every` loop (l. 2–18) reacts to each occurrence of `ENQUEUE`, creating and enqueuing the requested command, as illustrated in Figure 10:

- The initial state (box `0`) assumes a pre-existing neutral `ITEM` in the root of the `waiting` field (Ⓓ).
- Lines 4–11 assign a new subtree to the `tmp` field (box `1`) with a new neutral `ITEM` (Ⓔ, l. 5–6) linked to the set of commands to `MOVE` the turtle (Ⓕ, l. 7–11).
- Lines 15–16 move the already `waiting` commands (Ⓖ) to the tail of `tmp`, in the place of `NIL` (Ⓗ). The old location is automatically set to `NIL`. Note that we skip the neutral `ITEM` nodes of both `waiting` and `tmp` fields. This prevents the `waiting` root to become `NIL` and awake the `ROOT` node (*CODE-4*: 8) before we finish the enqueuing operation.
- Line 17 moves the `tmp` subtree (Ⓔ) back to the `waiting` field, releasing the abandoned neutral `ITEM` (Ⓓ), and notifying the `ROOT` node that the queue is no longer empty. The `tmp` field is automatically set to `NIL`. Note that the new `waiting` subtree preserves a neutral `ITEM` for subsequent enqueue operations.

4. RELATED WORK

Traversing data reactively in an imperative language requires dealing with concurrent updates. While attempts are made to make to this process more transparent, performance concerns ultimately require the programmer to specify behavior explicitly. In [4], one-way dataflow constraints are used to track updates in data structures in a reactive imperative language. In it, classes are annotated with constraint handler functions, recursively called in the event of value updates. Another approach, focused on incremental computation, is self-adjusting computation [1], using a combination of dynamic dependency graphs and memoization. Keeping track of dependencies incurs significant overhead: traceable data types [2] are one way to mitigating this issue, letting

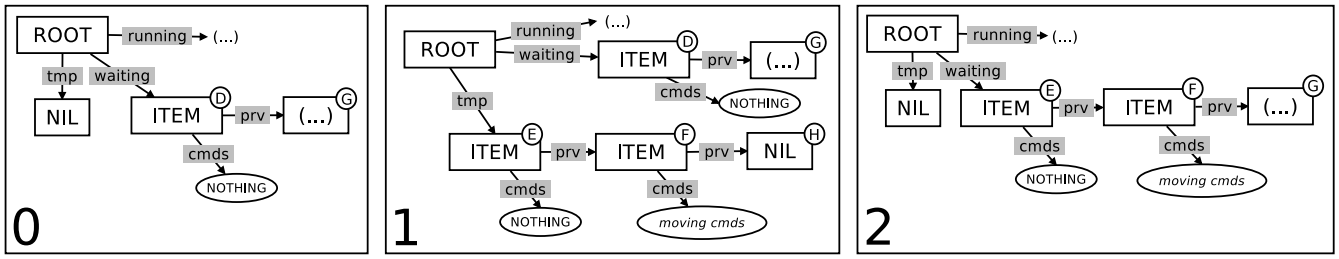

Figure 10: Enqueuing new commands.

the programmer control the granularity of tracked data. In Céu, tracking pointer updates is explicit (and enforced), and recursive data fields have move semantics designed for static memory management based on pools.

Introducing a control structure specifically for traversing recursive data is unusual, but parallels can be made to other languages. Generators trace their history back to CLU [11], but their implementation was stackless and therefore did not support recursion. Icon [5] allows yielding through recursive functions, but delegation is explicit via the `suspend` keyword. In [5], Icon generators are used for implementing goal-directed programming via "control backtracking" similarly to selector nodes presented in Section 3.1. Python originally introduced CLU-like stackless generators in version 2.2 [18], but those were later internally promoted to coroutines in order to support recursion [16]. Finally, in version 3.3, a form of delegation was introduced. In contrast, Lua [8] has first-class stackful coroutines: generator functions have to be constructed by wrapping coroutine objects, but there is no need for explicit delegation in recursive calls. Céu builds its higher-level `traverse` construct on organisms, which are also a primitive for cooperative multitasking, supporting recursion and concurrency transparently. This allows composing parallel traversals naturally; in contrast, traversing using multiple coroutines in Lua or Python requires an explicit coroutine scheduler.

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 that handles each recursive step in a separate organism (Céu's abstraction mechanism), allowing concurrent traversal while preserving the language's safety properties.

However, combining concurrency and safety while traversing a mutable recursive data structure is not straightforward. For instance, composing parallel constructs during recursion requires that each recursive step spawns new lines of execution. Also, ensuring that a traversal does not manipulate stale subtrees (originated from mutations) requires watching nodes to abort it. Additionally, by presenting a control construct that is tied to a data structure, we can ensure bounded execution time, in line with the philosophy of Céu. By dealing with these concerns internally in the `traverse` statement, we make reactive traversal as easy to perform correctly as a recursive function call.

We impose restrictions to the kinds of structures that can be represented with recursive data types in Céu. On the one hand, the requirement of a tree hierarchy with move semantics for assignments demands care in the design of algorithms manipulating these structures. On the other hand, these restrictions enable static memory management with deterministic deallocation. Still, we do not feel that the restrictions are prohibitively limiting. For instance, persistent data structures in functional languages [12] operate under tighter design constraints.

Possibilities for future work are the introduction of type arguments for `data` declarations, and investigating possibilities for relaxing Céu pointer and reference restrictions without sacrificing safety.

6. REFERENCES

[1] U. A. Acar et al. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.*, 32(1):3:1–3:53, Nov. 2009.

[2] U. A. Acar et al. Traceable data types for self-adjusting computation. In *Proceedings of PLDI '10*, PLDI '10, pages 483–496. ACM, 2010.

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

[4] C. Demetrescu et al. Reactive imperative programming with dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 37(1):3:1–3:53, Nov. 2014.

[5] R. E. Griswold, D. R. Hanson, and J. T. Korb. Generators in Icon. *ACM Trans. Program. Lang. Syst.*, 3(2):144–161, Apr. 1981.

[6] C. Hecker. My liner notes for Spore. `http://chrishecker.com/My_liner_notes_for_spore`, 2009. Accessed: 2015-07-30.

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

[8] R. Ierusalimschy. *Programming in Lua, Third Edition*. Lua.Org, 3rd edition, 2013.

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

[10] M. N. Krohn et al. Events can make sense. In *USENIX Annual Technical Conference*, pages 87–100, 2007.

[11] B. Liskov. A history of CLU. *SIGPLAN Not.*, 28(3):133–147, Mar. 1993.

[12] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[13] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., 1980.

[14] F. Sant'Anna et al. Safe system-level concurrency on resource-constrained nodes. In *SenSys'13*. ACM, 2013.

[15] F. Sant'Anna et al. Structured synchronous reactive programming with Céu. In *Modularity'15*, 2015.

[16] N. Schemenauer et al. PEP 255. https://www.python.org/dev/peps/pep-0255/. Accessed: 2015-08-03.

[17] J. Slaney and S. Thiébaux. Blocks world revisited. *Artif. Intell.*, 125(1-2):119–153, Jan. 2001.

[18] G. van Rossum and P. J. Eby. PEP 342. https://www.python.org/dev/peps/pep-0342/. Accessed: 2015-08-03.