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 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 offered by a language for constructing data types and values have a direct impact on the nature of algorithms that programmers of that language will write. The design of such facilities, on its turn, must take into account the constraints imposed by the rest of the language. For example, functional languages aim for referential transparency, and for this reason they typically present recursive data types as constructors for immutable data structures. One must then take care when writing algorithms using those structures to avoid excessive memory copying. A means to do that is to use data structures that alleviate the lack of random access to components and in-place modification, such as the ones presented by Okasaki in [1].

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 [3] 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. Being an imperative language, CÉU features mutable data. At the same time, however, its design promotes static memory management through lexically-scoped memory pools, and safety guarantees for concurrent pointer manipulation through orthogonal abortion [4]. These features preclude the availability of general mutable records such as C-style structs with arbitrary pointers. Recursive data types in CÉU must, therefore, respect the language's memory management guarantees and their use must cope with the synchronous concurrency model.

The solution to this problem is twofold, with data and control aspects to it. For data management, we introduce to the language a `data` type construct with a set of restrictions on instance constructors and assignment semantics that makes static memory management possible. For handling reactive control, 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.

Following the presentation of these constructs and their design, we showcase their use by discussing three applications in the domains of incremental computation and control-oriented DSLs that contain reactive and recursive behavior at the same time.

2. CÉU CONSTRUCTS
In this section, we present the constructs added to CÉU to support recursive data types with reactive traversal. We begin, though, with a short introduction to present the general flavor of the language.

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

```
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.

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 [3].

2.1 Recursive Data Types

The data construct in CÉU provides a safer alternative to C's struct, union, and enum definitions. A data entry declares either a non-recursive structure containing a set of mutable fields or a tagged union. A tagged union consists of a set of tag declarations, each of which may be a bare tag or contain mutable fields. If any of the tag declarations refers to the data type being declared, we have a recursive data type. In this case, the first tag of the tagged union must be a bare tag, and it will act as the union's null type: in CÉU, every recursive data type is an option type.

Figure 2 illustrates the recursive List data type, declared as a tagged union. The first tag, NIL (line 2), represents the empty list and is the union's null type. The second tag, CONS, holds a value in its field head and a pointer to the rest of the list in the field tail (lines 4–7).

All memory allocated by CÉU constructs is managed by lexically-scoped memory pools. The pool keyword declares a memory pool of a given size and a reference to a root object. In line 11, we declare a pool of List objects of size 1, identified by root reference lst1, scoped by the do block in lines 10–19. The declaration also implicitly initializes the root to the null tag of the associated data type (i.e., List.NIL).

Then, in lines 12–15, we use the =new construct, which performs allocation and assignment: it attempts to dynamically allocate a list of three elements (using three List.CONS constructors in the assignment *r-value*), inferring the destination memory pool based on the assignment's *l-value* (i.e. lst1).

```
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] lst1;
12     lst1 =new List.CONS(10,
13                List.CONS(20,
14                 List.CONS(30,
15                  List.NIL())));
16     _printf("%d, %d\n", lst1:CONS.head,
17                    lst1:CONS.tail:NIL);
18        // prints 10, 1
19  end
20
21  do
22     pool List[] lst2;
23     lst2 =new List.CONS(10,
24                List.CONS(20,
25                 List.CONS(30,
26                  List.NIL())));
27     lst2:CONS.tail =new List.CONS(50, List.NIL());
28     _printf("%d\n", lst2:CONS.tail:CONS.head);
29        // prints 50 (20 and 30 have been freed)
30  end
```

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

Since the pool has size 1, only the allocation of first element succeeds, with the failed allocations returning the null tag for this type (i.e., List.NIL). The print command (line 16) outputs "10, 1": the head of the first element (the operator ':' is equivalent to C's '->') and a true value for the NIL check of the second element.

In the second block (lines 21–30), we declare the lst2 pool with an unbounded memory limit (i.e., List[] in line 22). Now, the three-element allocation succeeds (lines 23–26). Then, we mutate the tail of the first element to point to a newly allocated element in the same pool, which also succeeds (line 27). The print command (line 28) outputs "50", displaying the head of the new second element. In the moment of the mutation, the old subtree (containing values "20" and "30") is completely removed from memory. Finally, the end of the block (line 30) 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: they must represent tree-like structures. Elements in different pools cannot be mixed; and 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

```
1  pool List[3] l = new List.CONS(10,
2                       List.CONS(20,
3                        List.CONS(30,
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);
16       // prints 60
```

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

```
1  pool List[] l = <...>;  // 10, 20, 30
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          escape 0;
14       end
15    end;
16 _printf("sum = %d\n", sum);
```

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

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.

We begin by showing the flavor of the construct through an example. The code 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`. When used without event control mechanisms, as in this simple example, a `traverse` block is equivalent to an anonymous closure called recursively.

The `traverse` construct does not simply amount to an anonymous recursive block, however. It is designed to take into account the event system and memory management discipline of the language. As such, it is an abstraction defined in terms of more fundamental Céu features: *organisms*, which are objects with their own parallel trail of execution, akin to Simula objects; and orthogonal abortion, which handles cancellation of trails maintaining memory consistency [4].

Three aspects make `traverse` fundamentally different from an anonymous recursive function. First, each `traverse` call spawns a new anonymous organism, launching a new parallel trail of execution (as opposed to stacking a new frame in the current trail). Second, traversal is declared in terms of a specific memory pool. Therefore, for bounded pools (e.g., `List[3] l`), we can infer at compile time the maximum traversal depth. Third, 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.

To illustrate these differences, Figure 4 extends the body of the previous example with reactive behavior. For each recursive iteration, `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 [4]. This ensures that if side effects occurring in parallel affect the pointed object, no code uses stale pointers 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.

The `traverse` construct allows us to enforce bounded execution time, by performing a limited number of steps, each of them a separate synchronous reaction. This can be asserted by verifying that the structure of the recursive steps converge to the base cases, or simply by using a bounded memory pools, which allows us to limit the maximum number of steps to the size of the pool. Enforcing execution limits is an important requirement for constrained and real-time embedded systems, which is the original application domain of Céu [3].

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 [?], and extend them with parallel compositions. Then, we show a *Logo Turtle* [?] that can execute commands in parallel (e.g., `move` and `rotate`) and also incorporates a dynamic queue of commands issued concurrently with the running program. Finally, we implement *Gray code* generation [?] to illustrate how `traverse` can also be used for more general recursive algorithms without an associated data type.

3.1 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

```
data BTree with
   tag NIL;
or
   tag SEQ with
      var BTree first;
      var BTree second;
   end
or
   tag SEL with
      var BTree first;
      var BTree second;
   end
or
   tag LEAF with
      var Leaf leaf;
   end
end
```

Figure 5: A standard behavior tree with *sequence* and *selector* composite nodes.

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.

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

3.2 Logo Turtle

Our second example is an interpreter for a simple variant of the classic Logo turtle-graphics interpreter [2], which extends the Logo paradigm with a CÉU-like parallel execution construct. In our variant, we can instruct the turtle to move and rotate in parallel, tracing curves. We declare a data type which defines our abstract syntax, with each tag representing one of the supported Logo commands. A tree of nodes represents a program, and the interpreter is implemented as a traversal of this tree. The aim of this example is to demonstrate parallel traversal.

Figure 7 presents the data type `Command`, which specifies the abstract syntax of our Logo variant. As in traditional Logo, commands can be listed in sequence to be executed one after the other (represented through a chain of `SEQ` nodes), and commands can be repeated a number of times (denoted through a `REPEAT` node). Our variant includes `MOVE` and `ROTATE` nodes to move the turtle, but these are specified differently from traditional Logo: here, they take as arguments the speed at which they should affect the turtle. For example, a `Command.MOVE(50)` node directs the turtle to move at the speed of 50 pixels per second, indefinitely. The only way to make the turtle stop moving or rotating is through two CÉU-like extensions added to our Logo variant: `AWAIT` and `PAROR`. `AWAIT` simply awaits a given number of milliseconds. `PAROR`, modeled after the CÉU construct `par/or`, launches two commands in parallel, and aborts both of them as soon as one of them finishes. For example, the following construct would make the turtle move along a semicircle:

```
class BTreeInterpreter with
   pool BTree[]& btree;
do
   var int ret =
      traverse t in btree do
         if t:SEQ then
             var int ok = traverse t:SEQ.first;
             if ok == 0 then
                 escape ok;
             end
             ok = traverse t:SEQ.second;
             escape ok;
         else/if t:SEL then
             var int ok = traverse t:SEL.first;
             if ok != 0 then
                 escape ok;
             end
             ok = traverse t:SEL.second;
             escape ok;
         else/if t:LEAF then
            var int ret =
                do LeafHandler(t:LEAF.leaf);
            escape ret;
         end
      end;
   escape ret;
end

pool BTree[] btree =
   new BTree.SELECTOR(
      BTree.SEQ(
         BTree.LEAF(Leaf.SENSEONTABLE(3)),
         BTree.SEQ(
            BTree.LEAF(Leaf.MOVEBLOCKTOBLOCK(2,3,1)),
            BTree.LEAF(Leaf.MOVEFROMTABLE(3,2))
         )
      ),
      BTree.SEQ(
         BTree.LEAF(Leaf.MOVETOTABLE(3,2)),
         BTree.SEQ(
            BTree.LEAF(Leaf.MOVEFROMTABLE(2,1)),
            BTree.LEAF(Leaf.MOVEFROMTABLE(3,2))
         )
      )
   );

var int ret = do BTreeTraverse(btree);

_printf("ret = %d\n", ret);     // prints "3"
```

Figure 6: A straightforward interpreter for the standard behavior tree of Figure 5 and a blocksworld tree to execute.

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

Figure 7: DSL for a Logo turtle.

```
Command.PAROR(
   Command.AWAIT(1000),
   Command.PAROR(
      Command.MOVE(50),
      Command.ROTATE(180)))
```

Figure 8 depicts the interpreter. It is implemented as the `Interpreter` organism (declared with the `class` keyword). It holds as attributes a reference to the AST of commands (`cmds`, line 2) and a reference to a `Turtle` object which implements the UI. The execution body of the organism contains the `traverse` construct which runs the interpreter (lines 5–35).

We have then a test for each kind of tag in the data type. In lines 7–9, `SEQ` is handled by traversing each of its child commands, in sequence: the second invocation of `traverse` in that block (line 9) only runs after the first one (line 8) finishes. In lines 11–14, `REPEAT` is handled by traversing its command the specified number of times.

`MOVE` is handled in lines 16–18 by spawning a new organism called `TurtleMove`, which launches a separate trail of execution. The implementation of `TurtleMove` (not shown) updates the coordinates of the `turtle` instance it got as a parameter in its constructor. The implementation of `ROTATE` (lines 20–22) is similar.

In lines 24–25, `AWAIT` is implemented by simply causing the current trail of execution of the interpreter to await the given amount of time. Finally, `PAROR` (lines 27–32) 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 will be aborted.

```
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:SEQ then
8              traverse cmd:SEQ.one;
9              traverse cmd:SEQ.two;
10
11          else/if cmd:REPEAT then
12             loop i in cmd:REPEAT.times do
13                traverse cmd:REPEAT.command;
14             end
15
16          else/if cmd:MOVE then
17             do TurtleMove(turtle,
18                           cmd:MOVE.pixels);
19
20          else/if cmd:ROTATE then
21             do TurtleRotate(turtle,
22                             cmd:ROTATE.angle);
23
24          else/if cmd:AWAIT then
25             await (cmd:AWAIT.ms) ms;
26
27          else/if cmd:PAROR then
28             par/or do
29                traverse cmd:PAROR.one;
30             with
31                traverse cmd:PAROR.two;
32             end
33          end
34       end
35    end
36 end
```

Figure 8: The turtle interpreter.

Note that the entire interpreter block is surrounded by a `watching` construct (line 6). 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 being interpreted is mutated by code running in another trail.

3.2.1 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-1*): `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-1*: 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-2* 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 Interpreter class of Figure 8 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-3*. The external input event ENQUEUE (line 1) accepts *move* and *rotate* commands with an associated velocity and time (i.e., 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-2*) before we finish the enqueuing operation. The old location for the moved commands is automatically set to NIL (mark *(c)*).

```
var int[4] bits = [0, 0, 0, 0];

par/or do
   every VISIT do
      _printf("( ");
      loop i in $$bits do // $$ is the array size
         _printf("%d ", bits[i]);
      end
      _printf(")\n");
   end
with
   traverse idx in [$$bits] do
      if idx == $$bits then
         await NEXT;
      else
         traverse idx + 1;
         bits[idx] = 1 - bits[idx];
         traverse idx + 1;
      end
   end
end
```

Figure 12: Generator for 4-bit Gray code.

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

3.3 Gray Code Generation

4. RELATED WORK

...

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

```
data Queue with
  tag NIL;
or
  tag ROOT with
    var Queue running;
    var Queue waiting;
    var Queue tmp;
  end
or
  tag ITEM with
    var Command cmds;
    var Queue   prv;
  end
end









.
```

CODE-1: The `Queue` data type

```
traverse qu in queue do
  watching qu do
    if qu:ROOT then
      loop do
        par/and do
          traverse qu:ROOT.running;
        with
          await qu:ROOT.waiting;
        end
        qu:ROOT.running =
          qu:ROOT.waiting;
        qu:ROOT.waiting =
          new Queue.ITEM(
               Command.NOTHING(),
               Queue.NIL());
      end
    else/if qu:ITEM then
      traverse qu:ITEM.prv;
      do Interpreter(turtle,
                     qu:ITEM.cmds);
    end
  end
end
```

CODE-2: Queue traversal

```
input (char*,int,int) ENQUEUE;
every (cmd,vel,time) in ENQUEUE do
  if _strcmp(cmd,"move")==0 then
    queue.ROOT.tmp =
      new Queue.ITEM(
           Command.PAROR(
            Command.MOVE(vel),
             Command.AWAIT(time)),
            Queue.NIL());
  else/if _strcmp(cmd,"rotate")==0 then
    <...> // analogous to the MOVE above
  end
  queue.ROOT.tmp.ITEM.prv =
    queue.ROOT.waiting.ITEM.prv;
  queue.ROOT.waiting = queue.ROOT.tmp;
end






.
```

CODE-3: Command enqueuing

Figure 9: Queue extension for the Turtle DSL of Figures 7 and 8.

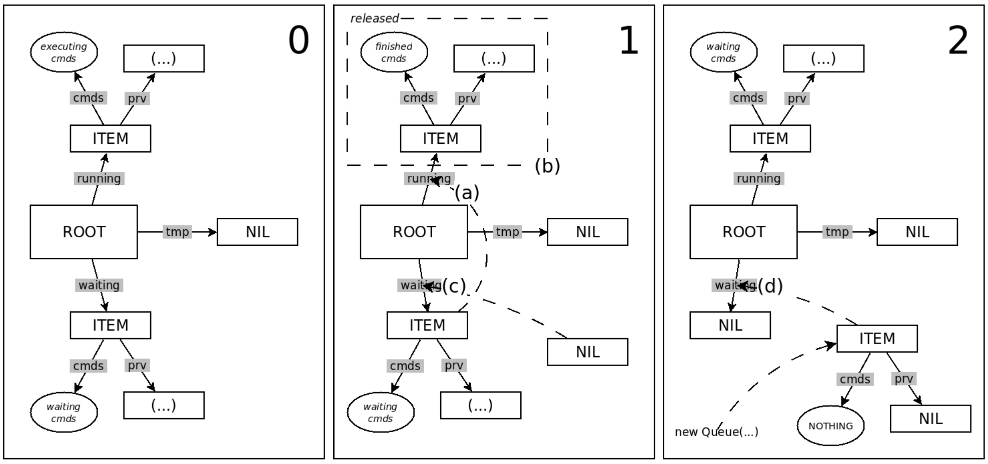

Figure 10: Swapping `waiting` and `running` commands.

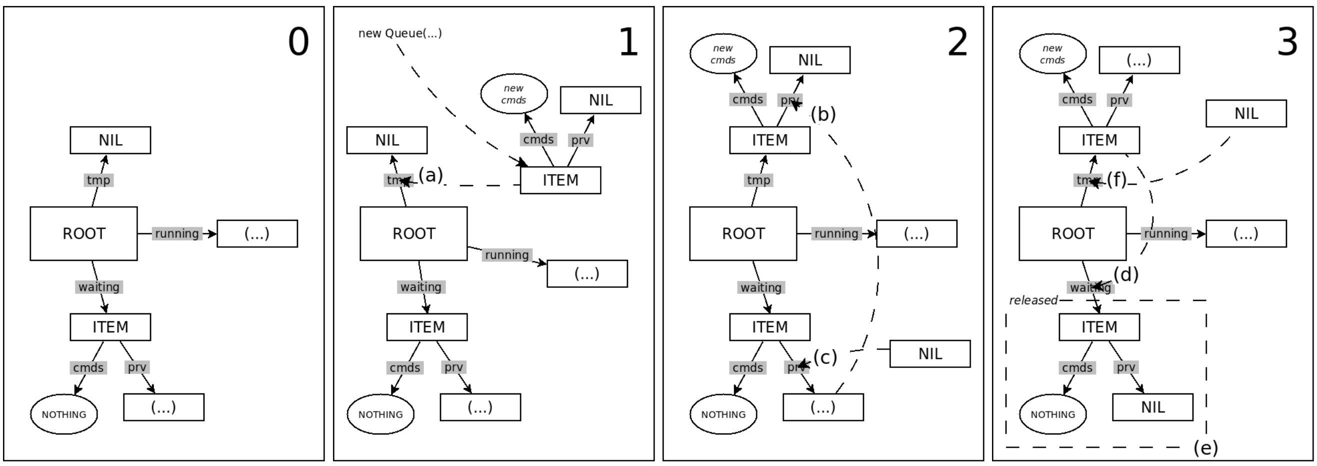

Figure 11: Enqueuing new commands.

fields requires care in the design of algorithms manipulating these structures, as illustrated in Section 3.2.1. 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.

Limitations:

- high-order programming

6. REFERENCES

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

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

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

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