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. The design of such facilities must take into account the constraints imposed by the rest of the language. As an example, the aim for referential transparency in functional languages enforces data structures to be immutable. Under these constraints, one must then avoid excessive memory copying through specialized algorithms [5].

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 [7, 8] 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, it promotes static memory management through lexically-scoped memory pools, and safe pointer manipulation even under concurrent trails with orthogonal abortion [8]. These features 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 `data` type construct with a set of restrictions on constructors and assignments that makes static memory management possible. For handling reactive control, we propose a structured mechanism that can 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.

After we present the design of these constructs, we discuss three applications in the domains of incremental computation and control-oriented DSLs. The applications contain reactive and recursive behavior at the same time and showcase the expressiveness of the proposed constructs. TODO: related/conclusion

2. CÉU CONSTRUCTS

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

First we present the general flavor of the language with an introductory example. The code 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 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

```
input void RESET;  // declares an external event
var int v = 0;     // variable shared by the trails
par do
   loop do         // 1st trail
      await 1s;
      v = v + 1;
      _printf("v = %d\n", v);
   end
with
   loop do         // 2nd trail
      await RESET;
      v = 0;
   end
end
```

Figure 1: Introductory example in Céu.

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 [7].

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), holds a value in the field `int head` and the rest of the list in the recursive field `List tail`. The `pool` keyword declares a memory pool of a given size and a (implicit) reference to its root object. All memory allocated by CÉU constructs is managed by lexically-scoped memory pools.

In the first block of the example (lines 7–16), we declare a pool of `List` objects of size 1 identified by the root reference `lst1` (line 8), which has the scope limited by the `do-end` block (lines 7–16). The declaration also implicitly initializes the root to the null tag of the associated data type (i.e., `List.NIL`). Then, we use the `=new` construct (lines 9–12), which performs allocation and assignment: 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 allocations returning the null tag (i.e., `List.NIL`). The print command (lines 13–14) 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. Finally, the end of the block (line 16) deallocates the pool along with all of its elements.

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.

[1]For the sake of brevity, we will omit the data type prefix for tags (e.g., `List.CONS` becomes `CONS`), given that no examples have clashes between identifiers.

```
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).

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

CÉU introduces a structured mechanism to traverse recursive data types. The `traverse` construct integrates well with the synchronous execution model, supporting nested control compositions, such as `await` and all `par` variations. It also preserves explicit lexical scopes with static memory management.

The 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 `l`. The `escape` statement (lines 6 and 9) returns a value to be assigned to the `sum` (line 3). A `NIL` list has `sum=0` (lines 5–6). A `CONS` list needs to calculate the `sum` of its tail recursively, invoking `traverse` again (line 8), which will create a nested instance of the enclosing `traverse` block (lines 4–11), now with `e` pointing to `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.

However, the `traverse` construct does not simply amount to an anonymous recursive block. 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 [8].

Three aspects make `traverse` fundamentally different from

```
pool List[3] l = <...>;  // [10, 20, 30]

var int sum =
   traverse e in l do
      if e:NIL then
         escape 0;
      else
         var int sum_tail = traverse e:CONS.tail;
         escape sum_tail + e:CONS.head;
      end
   end;

_printf("sum = %d\n", sum);
      // prints 60
```

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

```
1  pool List[3] l = <...>;  // [10, 20, 30]
2
3  var int sum =
4     traverse e in l 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
```

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

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 9) and awaits 1 second before traversing the `tail` (lines 10–11). In Céu, all accesses to a pointer that cross `await` statements must be protected with an enclosing `watching` block [8], which aborts its nested block if the referred object is released from memory. This ensures that if side effects occurring in parallel affect the pointed object, no code uses stale pointers 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).

Céu can also ensure termination for `traverse` by verifying that the structure of the recursive steps converge to the base cases, or by limiting the maximum number of recursive invocations based on the size of the data type pool. If a pool is limited to at most N elements, then you can have at most N alive `traverse` bodies. Enforcing execution limits is an important requirement for constrained and real-time embedded systems, which is the original application domain of Céu [7].

3. APPLICATIONS

In this section, we present two 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 [3] [1]. Then, we show a *Logo Turtle* [6] 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.

TODO: separate turtle/queue

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 semantics of the sequence node can be understood as short-circuit evaluation of an ‘and’; the `SEQ` node evaluates its left subtree until it finishes, and if it finishes successfully, evaluates its right subtree until it finishes. The semantics of the selection node can be understood as short-circuit evaluation of an ‘or’; the `SEL` node evaluates its left subtree until it finishes, and if it did not finish successfully, evaluates 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 for authoring game AI.

Because the evaluation of each tree extends across many frames, and there are sufficiently many creatures that it is infeasible for each creature to have its own thread, writing behavior tree nodes and leaves in other languages is an exercise in stack-ripped event-driven programming [4]. By lowering the barrier to writing custom nodes and leaves, Céu makes behavior trees more usable.

The leaves of a behavior tree in a blocks domain might include sensors that quickly either succeed or fail, as well as actuators that move blocks and unconditionally succeed.

In this case, we have somehow pinned down the situation to two possibilities[2]. We use a `SEL` node and a sensor leaf to decide which strategy is appropriate, and `SEQ` nodes and actuator leaves to execute it. This illustrates how the behavior tree can exhibit goal-directed behavior. The goal-directedness is not usually automatic; rather, behavior trees are authored. The `traverse` feature allows behavior to be authored by editing a script that is interpreted at runtime.

The tree in Figure 7 is based on output from Contingent-FF [2], and the blocks domain is one of its benchmark problems. The `traverse` feature and integration with C allows Céu to to consume the output of CPU-intensive algorithms at runtime.

[2]If 3 is on the table, then we should move 2 from 3 to 1, and then 3 from the table to 2, achieving a 123 stack. But if 3 is not on the table, we should move 3 from 2 to the table, and then move 2 from the table to 1, and then 3 from the table to 2, also achieving a 123 stack.

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

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

```
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
```

Figure 6: A straightforward interpreter for the behavior tree of Figure 5.

```
pool BTree[] btree =
  new SEL(
         SEQ(
            LEAF(SENSE_ON_TABLE(3)),
            SEQ(
               LEAF(MOVE_B_TO_B(2, 3, 1)),
               LEAF(MOVE_FROM_T(3, 2)))),
         SEQ(
            LEAF(MOVE_TO_T(3, 2)),
            SEQ(
               LEAF(MOVE_FROM_T(2, 1)),
               LEAF(MOVE_FROM_T(3, 2)))));

var int ret = do BTreeInterpreter(btree);
```

Figure 7: An example blocks behavior tree.

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

Figure 8: DSL for a Logo turtle.

3.2 Logo Turtle

Our second example is an interpreter for a simple variant of the classic Logo turtle-graphics interpreter [6], 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 8 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:

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

Figure 9 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)

```
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 9: The turtle interpreter.

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.

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.

3.3 Dynamic Data Structures

All examples so far create a fixed tree that does not vary during traversal. Figure 10 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 12 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 12 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 9 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 13 (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)`).

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

...

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.

Limitations:

- high-order programming

6. REFERENCES

[1] 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].

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

[3] D. Isla. Handling complexity in the halo 2 ai. In *Game Developers Conference, San Francisco*, 2005.

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

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

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

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

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

```
data Queue with
  tag NIL ();
or
  tag ROOT (Queue running,
            Queue waiting,
            Queue tmp);
or
  tag ITEM (Command cmds,
            Queue prv);
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 ITEM(NOTHING(), 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 ITEM(
           PAROR(MOVE(vel), AWAIT(time)),
           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 10: Queue extension for the Turtle DSL of Figures 8 and 9.

```
pool List[10] lst;
var int id1 = <...>;

var int id2 = traverse e in lst do
                  var int id3 = traverse e:CONS.tail;
                  escape id1 + id3;
              end;
.
```

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

```
pool List[10] lst;
var int id1 = <...>;

class Body with
   pool Body[10]& bodies;
   var  Body&     parent;
   var  Main&     outer;
   pool List[10]* e;
do
   watching this.parent do
      var Body* body = spawn Body(this.bodies, &this,
                                  &this.outer,
                                  &lst:CONS.tail);
                    in this.bodies;
      var int id3 = await *body;
      escape this.outer.id1 + id3;
   end
   escape 0;
end
pool Body[10] bodies;

var Body* body = spawn Body(bodies,&this,&this,&lst)
                 in bodies;
var int id2 = await *body;
```

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

Figure 11: "Desugaring" of the `traverse` **construct.**

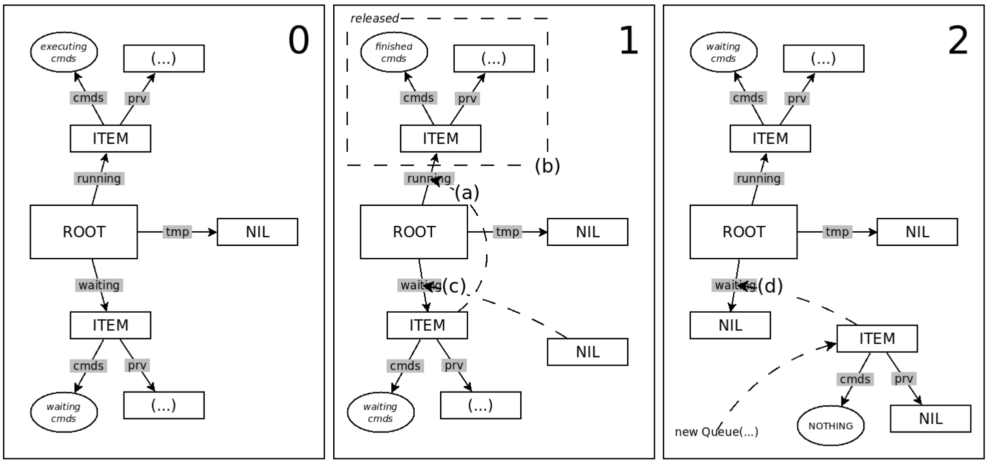

Figure 12: Swapping `waiting` and `running` commands.

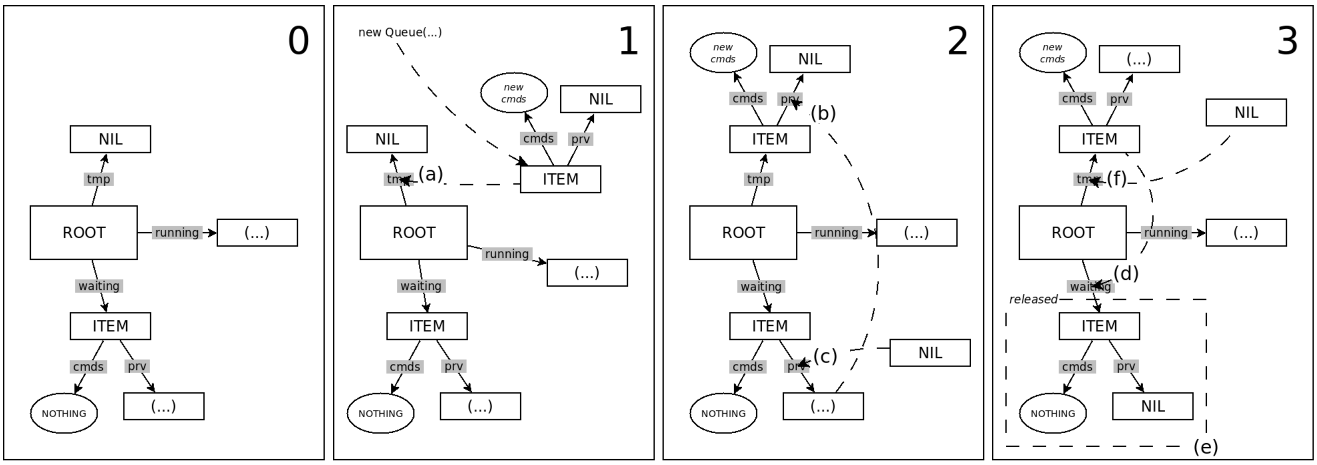

Figure 13: Enqueuing new commands.