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 of 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
Reactive Programming, Recursive Data Types, Structured Programming, Behavior Trees, Domain Specific Languages, Incremental Computation, Logo

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 [14].

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 [17, 18] 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. However, these features are incompatible with garbage-collected immutable data structures, as well as with general records with arbitrary pointers such as structs in C.

The solution to this problem is twofold, with data and control aspects. For data management, we introduce a restricted form of recursive data types that can describe 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 example1 in Figure 1 gives a general flavor of Céu. It first defines an input event RESET (line 1), a shared variable v (ln. 2), and starts two trails with the par construct (ln. 3-14): the first (ln. 4-8) increments variable v on every second and prints its value on screen; the second (ln. 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 awake

Figure 1: Introductory example in Céu.

1A screencast of all examples in the paper is available at https://vimeo.com/135297440.
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 \textit{await} statements [17].

CÉU relies on a source-to-source compiler that generates single-threaded code in C. The generated code has a negligible overhead in terms of memory and CPU usage in comparison to handcrafted event-driven code written directly in C [17].

\subsection{Recursive Data Types}

The \texttt{data} construct in CÉU provides a safer alternative to C’s \texttt{struct}, \texttt{union}, and \texttt{enum} definitions. All allocations are made in terms of a memory pool, and instances must form a tree structure with respect to the pool’s \textit{root element}.

Figure 2 illustrates the recursive \texttt{List} data type, declared as a tagged union (ln. 1–5). The first tag of a recursive type has a special meaning and represents the union’s \textit{null tag}. In the example, the tag \texttt{NIL} also represents an empty list (ln. 2). The second tag, \texttt{CONS}, (ln. 4), represents list nodes.

In the first block of the example (ln. 7–16), we declare a \texttt{pool} of \texttt{List} objects of size 1 (ln. 8). All pools have static memory management based on their enclosing lexical scope (ln. 7–16). The pool variable represents the root element, and is implicitly initialized to the \textit{null} tag of its data type, i.e., \texttt{lst1} receives \texttt{List\texttt{.NIL}} (ln. 8). Then, we use the \texttt{new} construct which performs allocation and assignment at the same time (ln. 9–12): it attempts to allocate a list of three elements (10, 20, and 30), and assigns the result to \texttt{lst1}, which is also the destination memory pool for the allocation (inferred from the \textit{l-value} in the assignment). Since the pool has size 1, only the allocation of first element succeeds, with the failed subtree allocation returning the \textit{null} tag (i.e., \texttt{List\texttt{.NIL}}). The print command (ln. 13–14) outputs “10, 1”: the \texttt{head} of the first element, and \texttt{true} in the \texttt{NIL} check for the second element. Finally, the end of the block (ln. 16) deallocates the pool along with all elements inside it.

In the second block (ln. 18–24), we declare the \texttt{lst2 pool} with an unbounded memory limit (i.e., \texttt{List\texttt{[\ldots]}} in line 19). Now, all allocations succeed (ln. 20).\footnote{To save space, in the next examples we omit the data type prefix in \texttt{tags} (e.g., \texttt{List\texttt{.CONS}} becomes \texttt{CONS}).} Then, we mutate the tail of the first element to point to a newly allocated element in the same pool, which also succeeds (ln. 21). At the moment of the mutation, the old subtree (containing values “20” and “30”) is completely removed from memory. The print command (ln. 22) outputs “50”, displaying the \texttt{head} of the new second element. Again, the end of the block (ln. 24) deallocates the pool along with all of its remaining elements.

In CÉU, recursive data types impose some restrictions. Elements in different pools cannot be mixed without copying. Types may contain arbitrary weak pointers, but weak pointers to CÉU structures must be observed via the \texttt{watching} construct, as they can be invalidated at any time (to be discussed in Section 2.2).

\subsection{Traversing Data Types}

CÉU introduces the \texttt{traverse} structured mechanism for traversing recursive data types incrementally. The code in

\begin{verbatim}
data List with
tag NIL();
or
tag CONS (int head, List tail);

end
do
pool List[] lst1;
lst1 = new List.CONS(10,
    List.CONS(20,
        List.CONS(30,
            List.NIL())));
_printf("%d, %d\n", lst1.CONS.head,
    lst1.CONS.tail.NIL);
    // prints 10, 1
end
do
pool List[] lst2;
lst2 = new CONS(10, CONS(20, CONS(30, NIL())));
_printf("%d\n", lst2.CONS.tail.CONS.head);
    // prints 50 (20 and 30 have been freed)

end
\end{verbatim}

\textbf{Figure 2:} A recursive \texttt{List} data type definition (ln. 1–5) with uses (ln. 7–16 and 18–24).

\begin{verbatim}
pool List[3] lst = <...>; // [10, 20, 30] var int sum =
    traverse e in lst do
        if e:NIL then escape 0;
        else
            var int sum_tail = traverse e:CONS.tail;
            escape sum_tail + e:CONS.head;
        end;
_printf("sum = %d\n", sum); // prints 60
\end{verbatim}

\textbf{Figure 3:} Calculating the \texttt{sum} of a list.

Figure 3 creates a list (ln. 1) and traverses it to calculate the \texttt{sum} of elements (ln. 3–11). The \texttt{traverse} block (ln. 4–11) starts with the element \texttt{e} pointing to the root of the list \texttt{lst}. The \texttt{escape} statements (ln. 6,9) return a value to the enclosing assignment to \texttt{sum} (ln. 3). The \texttt{sum} of a \texttt{NIL} list is \texttt{0} (ln. 5–6).\footnote{The operator ’::’, as in \texttt{e:NIL}, is equivalent to C’s ‘->’.} A \texttt{CONS} list needs to calculate the \texttt{sum} of its tail recursively, invoking \texttt{traverse} again (ln. 8), which creates a nested instance of the enclosing \texttt{traverse} block (ln. 4–11), now with \texttt{e} pointing to \texttt{e:CONS.tail}. Only after the recursive traversal of its subtree terminates the \texttt{CONS} clause adds its \texttt{head} and returns the sum (ln. 9).

When used without event-based control mechanisms, as in Figure 3, a \texttt{traverse} block is equivalent to an anonymous closure called recursively. However, \texttt{traverse} compiles with the event system and memory management discipline of CÉU. In fact, it is an abstraction defined in terms of a more fundamental concept, \textit{organisms} [18], which are objects with concurrent trails of execution (akin to Simula [3]). Figure 4 depicts the expansion of the \texttt{traverse} construct along with a non-trivial example.

The example in \texttt{CODE-1} of Figure 4 (\texttt{\(\square\)}) extends the body of the previous example in Figure 3 with reactive behavior. Now, for each recursive iteration, we print the current \texttt{head} and \texttt{await} 1 second before traversing the \texttt{tail} (ln. 10–12). Note that while nested iterations of \texttt{traverse} await
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.

1 second, all previous iterations are blocked, retaining their full state of execution. Note also that a parallel trail (ln. 21–24) remains reactive in the meantime, and can mutate the list at any time. We handle the issue of parallel mutations using the watching construct. The watching block aborts if the object being watched is released from memory [18]. Using watching e (ln. 9–14) we ensure that if element e is released from memory during await (ln. 11) due to a mutation in the list (ln. 22), we simply ignore the whole subtree and return 0 (ln. 15). Céu enforces at compile time that all accesses to pointers that cross await statements are protected with an enclosing watching block. This ensures that if concurrent side effects affect the pointed object, no code uses the stale pointer, because the whole block is aborted.

CODE-2 in Figure 4 (C2) is the equivalent expansion of C1 without the traverse construct. Because it contains concurrency constructs (i.e., await and watching), the body of the traverse construct (C2: 6–16) is abstracted in an organism of the Frame class (C2: 4–26), which is analogous to “stack frames” for subroutines in standard programming languages. In Céu, organisms also reside in lexically-scoped memory pools [18], hence, the pool of frames (C2: 29) is analogous to a runtime “call stack”. We limit the number of stack frames to match the exact maximum number of elements to traverse (C3: 1 and C2: 4, 29). To “call” the first traverse iteration, we dynamically spawn a Frame instance into the frames pool (C2: 31–32), which runs implicitly in parallel with the rest of the application (e.g., 36–39). Then, we immediately await the termination of this frame (C2: 33). Only after the whole traversal finishes that we acquire the sum and print it (C2: 33–34).

A Frame receives three arguments in the constructor (C2: 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 (C2: 31–32) receives the static pool of frames, a dummy scope organism attached to the current scope, and the original tree to traverse (C2: 5). The Frame constructor for recursive calls (C2: 17–19) receives the same pool of frames, another dummy scope as parent, and the subtree for the recursive invocation (C2: 12). The Frame body (C2: 8–27) watches the dummy organism attached to its parent scope and aborts if the parent terminates. Watching an object in the parent scope 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 (C2: 6–16 and C2: 9–25), with the exception of the recursive invocation explained above (C2: 16–19).

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 executes concurrently with other parts of the application. Also, each frame itself can execute arbitrary code in Céu with nested concurrent trails (to be illustrated in Section 3).
2. A traverse is attached to a lexically-scoped memory pool for a specific 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 [17]. In addition, when a recursive data structure goes out of scope, all associated stack frames are automatically aborted, given that the pool of frames expands to a narrower lexical scope (e.g., @: 1 vs. 29) [18].
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 (e.g., @: 22).

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

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. 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 [10, 7]. 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 and 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 extends across multiple game frames, specifying node behaviors in generic languages via event-driven programming becomes a challenge due to “stack ripping” [11]. 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 (ln. 1–9). The SEQ and SEL tags (ln. 4,6) are recursive and behave as described above. The LEAF tag (ln. 8) receives a reference to an opaque Leaf data type, which is defined externally and is specific to the application domain. The interpreter for behavior trees is abstracted in a class definition (ln. 11–35) and receives the tree to traverse as the single argument (ln. 11). The body acquires the return status of the traversal (ln. 12) and returns it as the final result (ln. 34). For the SEQ tag (ln. 14–20), we traverse the first subtree (ln. 15) and only if it succeeds, we traverse the second subtree (ln. 19). For the SEL tag (ln. 21–27), we traverse the first subtree (ln. 22) and only if it fails, we traverse the second subtree (ln. 26). Finally, the LEAF tag (ln. 28–31) delegates the behavior to another class, which does real work and is domain specific. The do class syntax (ln. 29–30) creates an anonymous and lexically scoped organism and awaits its termination to return the final status (ln. 31). The organism itself can contain any valid code in Céu (including parallel compositions) and executes for an arbitrary amount of time [18].

As an example of a domain, the blocks world is a classical planning domain in AI [19]. The tree in Figure 6 is based on the output from a Contingent-FF benchmark that extends the blocks domain with sensor actions [8]. We want to achieve an ABC stack and assume two decision possibili-
ties, as illustrated in the figure. We use a SEL node (ln. 3) with a sensor leaf (ln. 4) to decide which strategy is appropriate: If C is not sensed on top of the table, we first move it to the table (ln. 5). Then, in both situations, we stack B on top of A, and C on top of B (ln. 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 [15]. The aim of this example is to demonstrate parallel traversal: we can instruct the turtle to move and rotate at the same time to trace curves.

Figure 7 presents the Command data type (ln. 1–15), which specifies the abstract syntax of our Logo variant. As in traditional Logo, commands can execute in sequence through the SEQ tag (ln. 4), and can also repeat a number of times through the REPEAT tag (ln. 6). We extend the MOVE and ROTATE commands to take as arguments the speed at which they should affect the turtle (ln. 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: The AWAIT tag (ln. 12) simply awaits a given number of milliseconds. The PAROR (ln. 14) launches two commands in parallel, and aborts both as soon as one of them terminates. As an example, the program in lines 54–56 makes the turtle to move along a semicircle.

The interpreter for the commands is also abstracted in a class definition (ln. 17–51). It holds as attributes a reference to aQueue of pending commands, and aQueue on termination (ln. 8–9) represents a queue item and contains the actual commands to execute (as described in Figure 7: 1–15), and a par queue item pointing to an older item (i.e., the queue is in reverse order to facilitate insertion). 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 fields. Except when creating a new command, the tail field is always NIL.

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

```
data Command with
  tag NOTHING();
  or
  tag SEQ (Command first, Command second);
  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 first, Command second);
end

class CommandInterpreter (Turtle& turtle, Command[]& cmds)
do
  traverse cmd in cmds do
    watching cmd do
      if cmd:SEQ then
        traverse cmd:SEQ.first;
        traverse cmd:SEQ.second;
      else
        if cmd:REPEAT then
          loop i in cmd:REPEAT.times do
            traverse cmd:REPEAT.command;
          end
        else
          if cmd:MOVE then
            do TurtleMove(turtle, cmd:MOVE.pixels);
          else
            if cmd:ROTATE then
              do TurtleRotate(turtle, cmd:ROTATE.angle);
            else
              if cmd:AWait then
                await (cmd:AWait.ms) ms;
              else
                if cmd:PAROR then
                  par/or do
                    traverse cmd:PAROR.first;
                    with
                      traverse cmd:PAROR.second;
                  end
                end
              end
            end
          end
        end
      end
    end
  end

pool Command[]& cmds =
  new PAROR(
    AWAIT(1000),
    PAROR(MOVE(300), ROTATE(180)));

var Turtle turtle;
do CommandInterpreter(turtle, cmds);
```
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());
    else
      if qu:ITEM then
        traverse qu:ITEM.prv;
        CommandInterpreter(turtle, qu:ITEM.cmds);
      end
    end
  end
end

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

Figure 9: Swapping waiting and running commands.
- The initial state (box 0) assumes pre-existing running and waiting items.
- Lines 10–11 assign the waiting subtree (marked (⊙)) to the running field, releasing the old subtree (ซะ). Recursive data types in CEU have move semantics, hence, the waiting field is automatically set to the union null type (i.e., Queue.NIL).
- Lines 12–13 assign a new neutral ITEM with a dummy command NOTHING (box 1: ⃗) to the waiting field, completing the swapping operation.

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

Even though this example mutates the running field only after its traversal terminates (ln. 10–11), it is safe to do an arbitrary mutation at any time. Note that the compiler enforces the use of watching (ln. 2) which encloses the running turtle interpreter (ln. 17–18). Hence, if its enclosing ITEM (ln. 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 (ln. 1) accepts "move" and "rotate" strings with an associated velocity and time (i.e., "char*,int,int" arguments). The every loop (ln. 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).

Figure 10: Enqueuing new commands.
4. RELATED WORK

Traversing data reactively in an imperative language requires dealing with concurrent updates. Some attempts make this process more transparent, but performance concerns ultimately require the programmer to specify behavior explicitly. One way dataflow constraints were used to track updates in data structures in a reactive imperative language [4]. In it, classes are annotated with constraint handler functions which are recursively called on value updates. Another approach focuses on incremental computation and uses a combination of dynamic dependency graphs and memoization [1]. Keeping track of dependencies incurs significant overhead: traceable data types [2] mitigate this issue, letting the processor control the granularity of tracked data. In Céu, tracking pointer updates is explicit (and enforced), and the move semantics for recursive data fields propagates changes to tracked pointers automatically.

More universal approaches such as generators first appeared in CLU [12], but with a stackless implementation that did not support recursion. Icon [5] allows yielding through recursive functions, but delegation is explicit via the suspend keyword. Python originally introduced CLU-like stackless generators in version 2.2, but those were later internally promoted to coroutines in order to support recursion. Finally, in version 3.3, a form of delegation was introduced [16]. Lua [9] has a first-class stackful coroutines: generator functions have to be constructed by wrapping coroutine objects, but explicit delegation in recursive calls is not required. More recently, extensions to support asynchronous concurrent updates during traversal have been proposed for Dart [13] and Scala [6]. Like our work, the Scala extension is translated into the language’s core concurrency model. Céu builds its higher-level traverse construct on top of organisms, which also provides cooperative multitasking, but with transparent support for recursion and concurrency. Organisms allow composing parallel traversals more naturally in comparison to coroutines, which requires a specialized scheduler and does not impose safety guarantees.

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. Combining concurrency and safety while traversing recursive data structures is not straightforward, especially with support for mutation.

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. A traverse block is tied with its associated data structure, ensuring bounded execution time. By dealing with concurrency in a primitive control mechanism, we make reactive traversal as easy to perform correctly as a recursive function call. We impose some restrictions on the data structures that can be represented. The requirement of tree hierarchies with move semantics demands care when designing algorithms that manipulate them. In trade, 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 operate under similar constraints.

6. REFERENCES