# Reactive Traversal of Recursive Data Types

Francisco Sant'Anna - PUC-Rio
fsantanna@inf.puc-rio.br
*Hisham Muhammad* - PUC-Rio
hisham@inf.puc-rio.br
Johnicholas Hines - IDEXX Laboratories
johnicholas.hines@gmail.com

# Context

- Céu language - http://ceu-lang.org
- Imperative structured reactive programming
- Supports different event backends
  - SDL backend for game development
  - Arduino backend for embedded systems
    (can run in very constrained systems)

# A quick look at Céu

```
// declares an external event
input void RESET;

// variable shared by both trails
var int v = 0;
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
```

# Properties of Céu

- synchronous language
- static memory management
- parallel compositions with safe abortion
- bounded reaction time
- bounded memory usage

# Goal

- Add recursive data structures to Céu
  - lists, trees, etc.

# Goal

- Add recursive data structures to Céu
  - lists, trees, etc.
- Allow incremental traversal of these structures while keeping the language's properties
  - static memory management
  - parallel compositions with safe abortion
  - bounded reaction time
  - bounded memory usage

# Recursive data types in Céu

- *data* construct
- tagged unions

```
data List with
   tag NIL (); // 1st tag is null tag
or
   tag CONS (int head, List tail);
end
```

# Recursive data types in Céu

- allocated in pools

```
do
   pool List[3] lst1;
   lst1 = new List.CONS(10,
                List.CONS(20,
                 List.CONS(30,
                  List.NIL())));
end
```

*Céu property:* bounded memory usage

# Recursive data types in Céu

- failed allocations default to the null tag

```
do
   pool List[2] lst1;
   lst1 = new List.CONS(10,
                List.CONS(20,
                 List.CONS(30, // fails
                  List.NIL())));
   // produces List.CONS(10,
                 List.CONS(20,
                  List.NIL());
end
```

*Céu property:* bounded memory usage

# Recursive data types in Céu

- references have move semantics
  - can only represent tree-like structures
- may contain other weak pointers

```
do
   pool List[] lst2;
   lst2 = new CONS(10,CONS(20,NIL()));

   // this frees cons "20":
   lst2.CONS.tail = new CONS(50,NIL());

   _printf("%d\n",
           l2.CONS.tail.CONS.head);
   // prints 50
end
```

*Céu property:* static memory management

# How to traverse recursive *data*

# How to traverse recursive *data*

- Céu does not have recursive functions!

# How to traverse recursive *data*

- Céu does not have recursive functions!
- How to traverse a *data* structure
  - Ensuring bounded execution time
  - Allowing parallel compositions with safe abortion
  - Incrementally: we don't want the whole traversal to be a single reaction

# The *traverse* construct

- A control structure for traversing *data*

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

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

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

# The *traverse* construct

- Recursion limited by the size of the given pool

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

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

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

*Céu property:* bounded execution

# The *traverse* construct

- Recursion limited by the size of the given pool

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

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

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

*Céu property:* bounded execution

# The *traverse* construct

- Inner calls to *traverse* continue the traversal
- yields to other reactions

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

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

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

*Céu property:* incremental execution

# The *traverse* construct

- if the enclosing scope terminates, the traversal terminates

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

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

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

*Céu property:* safe abortion

# Example: a Logo turtle with "par/or"

```
data Command with
   tag NOTHING ();
or
   tag SEQ (Command first, Command second);
or
   tag MOVE (int speed);
or
   tag ROTATE (int speed);
or
   tag AWAIT (int ms);
or
   tag PAROR (Command left, Command right);
end
```

```
class CommandInterpreter (Turtle& turtle, Command[]& cmds)
do
   traverse cmd in cmds do

      if cmd:SEQ then
         traverse cmd:SEQ.first;
         traverse cmd:SEQ.second;

      else/if cmd:MOVE then
         do TurtleMove(turtle, cmd:MOVE.speed);

      else/if cmd:ROTATE then
         do TurtleRotate(turtle, cmd:ROTATE.speed);

      else/if cmd:AWAIT then
         await (cmd:AWAIT.ms) ms;

      else/if cmd:PAROR then
         par/or do
            traverse cmd:PAROR.left;
         with
            traverse cmd:PAROR.right;
         end

      end
   end
end
```

# An example "Logo+par/or" program

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

# How *traverse* works

- *data* is a semantic addition to the language
- *traverse* is "syntactic sugar":
  - implemented in terms of existing Céu constructs

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

par do
  var int sum = traverse e in lst do
    if e:NIL then
      escape 0;
    else
      watching e do
        _printf("%d\n", e:CONS.head);
        await 1s;
        var int s = traverse e:CONS.tail;
        escape s + e:CONS.head;
      end
      escape 0;
    end
  end;
  _printf("sum = %d\n", sum);
  // prints 60 (with no mutations)
with
  <...>
    lst.CONS.tail = NIL();
    // possible concurrent mutation
  <...>
end
```

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

par do
  var int sum = traverse e in lst do
    if e:NIL then
      escape 0;
    else
      watching e do
        _printf("%d\n", e:CONS.head);
        await 1s;
        var int s = traverse e:CONS.tail;
        escape s + e:CONS.head;
      end
      escape 0;
    end
  end;
  _printf("sum = %d\n", sum);
  // prints 60 (with no mutations)
with
  <...>
    lst.CONS.tail = NIL();
    // possible concurrent mutation
  <...>
end
```

```
var int sum = traverse e in lst do
    if e:NIL then
        escape 0;
    else
        watching e do
            _printf("%d\n", e:CONS.head);
            await 1s;
            var int s = traverse e:CONS.tail;
            escape s + e:CONS.head;
        end
        escape 0;
    end
end;
```

```
class Frame(List[3]* e) do
  if e:NIL then
    escape 0;
  else
    watching e do
      _printf("%d\n", e:CONS.head);
      await 1s;
      var int s = traverse e:CONS.tail;
      escape s + e:CONS.head;
    end
    escape 0;
  end
end;
pool Frame fp[3];
var Frame* f = spawn Frame(lst) in fp;
var int sum = await *f;
```

A *traverse* block is a Céu *organism*: a reactive object with its own trail of code

*Céu property:* incremental execution

```
class Frame(List[3]* e) do
  if e:NIL then
    escape 0;
  else
    watching e do
      _printf("%d\n", e:CONS.head);
      await 1s;
      var int s = traverse e:CONS.tail;
      escape s + e:CONS.head;
    end
    escape 0;
  end
end;
pool Frame fp[3];
var Frame* f = spawn Frame(lst) in fp;
var int sum = await *f;
```

An organism is always spawned from a pool

*Céu property:* bounded memory usage

```
class Frame(List[3]* e) do
  if e:NIL then
    escape 0;
  else
    watching e do
      _printf("%d\n", e:CONS.head);
      await 1s;
      var int s = traverse e:CONS.tail;
      escape s + e:CONS.head;
    end
    escape 0;
  end
end;
pool Frame fp[3];
var Frame* f = spawn Frame(lst) in fp;
var int sum = await *f;
```

Likewise, we have to expand the inner *traverse* into a *spawn*...

```
class Frame(Frame[3]& fp, List[3]* e) do
  if e:NIL then
    escape 0;
  else
    watching e do
      _printf("%d\n", e:CONS.head);
      await 1s;
      var int s = traverse e:CONS.tail;
      escape s + e:CONS.head;
    end
    escape 0;
  end
end;
pool Frame fp[3];
var Frame* f = spawn Frame(fp, lst) in fp;
var int sum = await *f;
```

For that, we'll need a reference to our frame pool...

```
class Frame(Frame[3]& fp, List[3]* e) do
  if e:NIL then
    escape 0;
  else
    watching e do
      _printf("%d\n", e:CONS.head);
      await 1s;
      do
        var Frame* f = spawn Frame(this.fp, e:CONS.tail) in this.fp;
        var int s = await *f;
      end
      escape s + e:CONS.head;
    end
    escape 0;
  end
end;
pool Frame fp[3];
var Frame* f = spawn Frame(fp, lst) in fp;
var int sum = await *fp;
```

…so we can spawn the next frame organisms for the traversal.

```
class Frame(Frame[3]& fp, List[3]* e) do
  if e:NIL then
    escape 0;
  else
    watching e do
      _printf("%d\n", e:CONS.head);
      await 1s;
      do
        var Frame* f = spawn Frame(this.fp, e:CONS.tail) in this.fp;
        var int s = await *f;
      end
      escape s + e:CONS.head;
    end
    escape 0;
  end
end;
pool Frame fp[3];
var Frame* f = spawn Frame(fp, lst) in fp;
var int sum = await *fp;
```

Finally, we also need to ensure that frame organisms are aborted if the scope of *traverse* terminates…

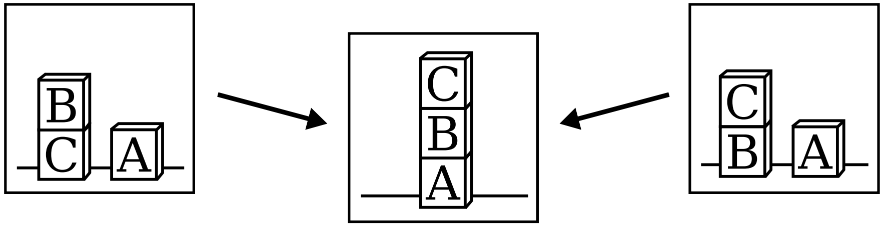
```
class Frame(Frame[3]& fs, _Dummy* enclosingScope, List[3]* e) do
  watching *this.enclosingScope do
    if e:NIL then
      escape 0;
    else
      watching e do
        _printf("%d\n", e:CONS.head);
        await 1s;
        do
          var _Dummy sc; // dummy variable in the scope of the caller
          var Frame* f = spawn Frame(this.fs,&sc,e:CONS.tail) in this.fs;
          var int s = await *f;
        end
        escape s + e:CONS.head;
      end
      escape 0;
    end
  end
end;
pool Frame fp[3];
var _Dummy sc; // dummy variable in the scope of the caller
var Frame* f = spawn Frame(fp, &sc, lst) in frames;
var int sum = await *f;
```

We guard the execution against the enclosing scope.

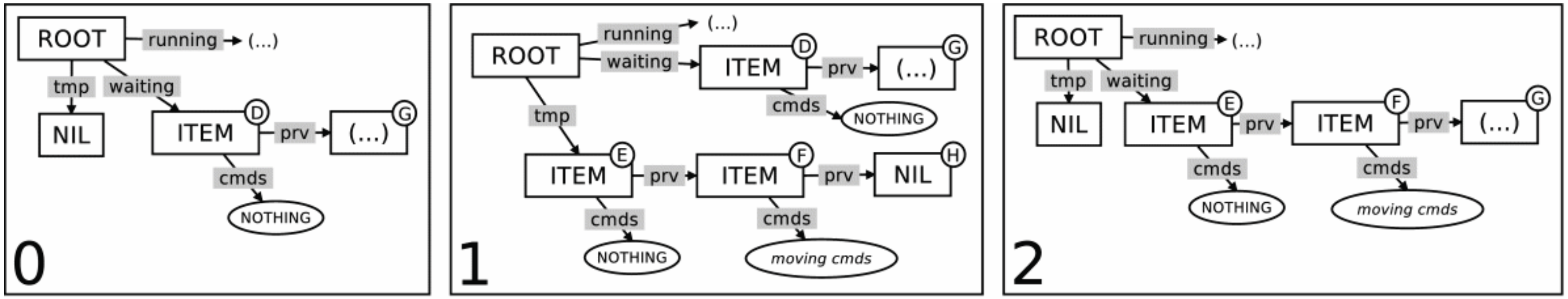
*Céu property:* safe abortion

# More examples in the paper

- Behavior Trees applied to blocks world




- Enqueuing commands in the Logo DSL (modifies the data structure as it traverses)



# Conclusions

- Recursive *data* structures in Céu
  - imperative synchronous reactive language
  - restrictions for static memory management with deterministic deallocation
- Incrementally *traverse* these structures while keeping the language's properties
  - bounded reaction time and memory usage
  - parallel compositions with safe abortion

# Thank you!

- For more on Céu:

  http://www.ceu-lang.org

- Our lab:

  LabLua - http://www.lua.inf.puc-rio.br

hisham@inf.puc-rio.br - @hisham_hm