A Memory-Bounded, Deterministic and Terminating Semantics for the Synchronous Programming Language Céu

Rodrigo C. M. Santos
PUC-Rio, Brazil
rsantos@inf.puc-rio.br

Guilherme F. Lima
PUC-Rio, Brazil
glima@inf.puc-rio.br

Francisco Sant'Anna
UERJ, Brazil
francisco@ime.uerj.br

Roberto Ierusalimschy
PUC-Rio, Brazil
roberto@inf.puc-rio.br

Edward H. Haeusler
PUC-Rio, Brazil
hermann@inf.puc-rio.br

Abstract

Céu is a synchronous programming language for embedded soft real-time systems. It focuses on control-flow safety features in the presence of shared-memory concurrency and abortion of lines of execution, while enforcing memory-bounded, deterministic, and terminating reactions to the environment. In this work, we present a small-step structural operational semantics for Céu and a proof that reactions have the properties enumerated above: that for a given arbitrary timeline of input events, multiple executions of the same program always react in bounded time and arrive at the same final finite memory state.

Keywords Determinism, Termination, Operational semantics, Synchronous languages

1 Introduction

Céu [19, 20] is a Esterel-based [10] programming language for embedded soft real-time systems that aims to offer a concurrent, safe, and expressive alternative to C with the characteristics that follow:

Reactive: code only executes in reactions to events.

Structured: programs use structured control mechanisms, such as `await` (to suspend a line of execution), and `par` (to combine multiple lines of execution).

Synchronous: reactions run atomically and to completion on each line of execution, i.e., there's no implicit preemption or real parallelism.

Structured reactive programming lets developers write code in direct style, recovering from the inversion of control imposed by event-driven execution [1, 15, 18]. Synchronous languages offer a simple run-to-completion execution model that enables deterministic execution and make formal reasoning tractable. For this reason, it has been successfully adopted in safety-critical real-time embedded systems [3].

Previous work in the context of embedded sensor networks evaluates the expressiveness of Céu in comparison to event-driven code in C and attests a reduction in source code size (around 25%) with a small increase in memory usage (around 5–10%) [20]. Céu has also been used in the context of multimedia systems [21] and games [**?**].

Céu inherits the synchronous and imperative mindset of Esterel but adopts a simpler semantics with fine-grained execution control. The list that follows summarizes the semantic peculiarities of Céu [19]:

- Fine-grained, intra-reaction deterministic execution, which makes Céu fully deterministic.
- Stack-based execution for internal events, which provides a limited but memory-bounded form of subroutines.
- Finalization mechanism for lines of execution, which makes abortion safe with regard to external resources.
- First-class timers with dedicated syntax, which provides automatic synchronization for multiple timers running simultaneously.

In this work, we present a formal small-step structural operational semantics for Céu and prove that it enforces memory-bounded, deterministic, and terminating reactions to the environment, i.e., that for a given arbitrary timeline of input events, multiple executions of the same program always react in bounded time and arrive at the same final finite memory state. Conceiving a formal semantics for Céu leads to a number of capabilities and outcomes as follows:

1. Understanding, finding corner cases, and stabilizing the language. After the semantics was complete and discussed in extent in our group, we found a critical bug in the order of execution of statements.
2. Explaining core aspects of the language in a reduced, precise, and unambiguous way. This is particularly important when comparing languages that are similar in the surface (e.g., Céu and Esterel).
3. Implementing the language. A small-step operational semantics describes an abstract machine that is close to a concrete implementation, which we concretized in Haskell for testing. Also, the current real-world implementation of the Céu scheduler is based on the formal semantics presented in this paper.

4. Proving properties for particular or all programs in the language. For instance, in this work, we prove that all programs in Céu are memory bounded, deterministic, and react in finite time.

The last item is particularly important in the context of constrained embedded systems:

Memory Boundedness: At compile time, we can ensure that the program fits in the device's restricted memory and that it will not grow unexpectedly during runtime.

Deterministic Execution: We can simulate an entire program execution providing an input history with the guarantee that it will always have the same behavior. This can be done in negligible time in a controlled development environment before deploying the application to the actual device (e.g., by multiple developers in standard desktops).

Terminating Reactions: Real-time applications must guarantee responses within specified deadlines. A terminating semantics enforces upper bounds for all reactions and guarantees that programs always progress with the time.

The rest of the paper is organized as follows: Section 2 is a review of the main characteristics of Céu based on previous work [19, 20], namely, deterministic event-driven execution, safe shared-memory concurrency, abortion and finalization for concurrent lines of execution, and first-class synchronized timers. Section 3 proposes a formal small-step operational semantics for Céu that encompasses all peculiarities of the language. Section 4 presents the proofs for the properties of memory boundedness, deterministic execution, and terminating reactions, which apply to all programs in Céu. Section 5 compares the semantics of Céu with related synchronous languages. Section 6 concludes the paper.

2 Céu

Céu [19, 20] is a synchronous reactive language in which programs evolve in a sequence of discrete reactions to external events. It is designed for control-intensive applications, supporting concurrent lines of execution, known as *trails*, and instantaneous broadcast communication through events. Computations within a reaction (such as expressions, assignments, and system calls) are also instantaneous considering the synchronous hypothesis [11]. Céu provides an `await` statement that blocks the current running trail allowing the program to execute its other trails; when all trails are blocked, the reaction terminates and control returns to the environment.

In Céu, every execution path within loops must contain at least a `break` or `await` statement to an external input event [6, 20]. This restriction, which is statically checked by the compiler, ensures that every reaction runs in bounded time, eventually terminating with all trails blocked in `await` statements. Céu has an additional restriction, which it shares with Esterel and synchronous languages in general [4]: computations that take a non-negligible time to run (e.g., cryptography or image processing algorithms) violate the zero-delay hypothesis, and thus cannot be directly implemented.

```
// Declarations
input ⟨type⟩ ⟨id⟩; // declares an external input event
event ⟨type⟩ ⟨id⟩; // declares an internal event
var   ⟨type⟩ ⟨id⟩; // declares a variable

// Event handling
⟨id⟩ = await ⟨id⟩; // awaits event and assigns the received value
emit ⟨id⟩(⟨expr⟩); // emits event passing a value

// Control flow
⟨stmt⟩ ; ⟨stmt⟩                                  // sequence
if ⟨expr⟩ then ⟨stmt⟩ else ⟨stmt⟩ end            // conditional
loop do ⟨stmt⟩ end                               // repetition
every ⟨id⟩ in ⟨id⟩ do ⟨stmt⟩ end                 // event iteration
finalize [⟨stmt⟩] with ⟨stmt⟩ end                // finalization

// Logical parallelism
par/or  do ⟨stmt⟩ with ⟨stmt⟩ end   // aborts if any side ends
par/and do ⟨stmt⟩ with ⟨stmt⟩ end   // terminates if all sides end

// Assignment & Integration with C
⟨id⟩ = ⟨expr⟩;    // assigns a value to a variable
_⟨id⟩(⟨exprs⟩)    // calls a C function (id starts with '_')
```

Listing 1. The concrete syntax of Céu.

Listing 1 shows a compact reference of Céu.

Listing 2 shows a complete example in Céu that toggles a LED whenever a radio packet is received, terminating on a button press always with the LED off. The program first declares the `BUTTON` and `RADIO_RECV` as input events (ln 1–2). Then, it uses a `par/or` composition to run two activities in parallel: a single-statement trail that waits for a button press before terminating (ln 4), and an endless loop that toggles the LED on and off on radio receives (ln 9–14). The `finalize` clause (ln 6–8) ensures that, no matter how its enclosing trail terminates, the LED will be unconditionally turned off (ln 7).

The `par/or` composition, which stands for a *parallel-or*, provides an orthogonal abortion mechanism [4] in which its composed trails do not know when and how they are aborted (i.e., abortion is external to them). The finalization mechanism extends orthogonal abortion to activities that use stateful resources from the environment (such as files and network handlers), as we discuss in Section 2.3.

In Céu, any identifier prefixed with an underscore (e.g., `_led`) is passed unchanged to the underlying C compiler. Therefore, access to C is straightforward and syntactically traceable. To ensure that programs operate under the synchronous hypothesis, the compiler environment should only provide access to C operations that can be assumed to be

instantaneous, such as non-blocking I/O and simple data structure accessors.[1]

```
1  input void BUTTON;
2  input void RADIO_RECV;
3  par/or do
4      await BUTTON;
5  with
6      finalize with
7          _led(0);
8      end
9      loop do
10         _led(1);
11         await RADIO_RECV;
12         _led(0);
13         await RADIO_RECV;
14     end
15 end
```

Listing 2. A Céu program that toggles a LED on receive, terminating on a button always with the LED off.

2.1 External and Internal Events

Céu defines time as a discrete sequence of reactions to unique external input events received from the environment. Each input event delimits a new logical unit of time that triggers an associated reaction. The life-cycle of a program in Céu can be summarized as follows [20]:

(i) The program initiates a "boot reaction" in a single trail (parallel constructs may create new trails).
(ii) Active trails execute until they await or terminate, one after another. This step is called a *reaction chain*, and always runs in bounded time.
(iii) When all trails are blocked, the program goes idle and the environment takes control.
(iv) On the occurrence of a new external input event, the environment awakes *all* trails awaiting that event, and the program goes back to step (ii).

A program must react to an event completely before handling the next one. By the synchronous hypothesis, the time the program spends in step (ii) is conceptually zero (in practice, negligible). Hence, from the point of view of the environment, the program is always idle on step (iii). In practice, if a new external input event occurs while a reaction executes, the event is saved on a queue, which effectively schedules it to be processed in a subsequent reaction.

2.1.1 External Events and Discrete Time

The sequential processing of external input events induces a discrete notion of time in Céu, as illustrated in Figure 1. The

[1] The actual implementation of Céu supports a command-line option that accepts a whitelist of library calls. If a program tries to use a function not in the list, the compiler raises an error.

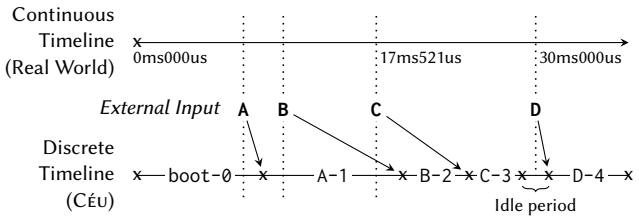

Figure 1. The discrete notion of time in Céu.

continuous timeline shows an absolute reference clock with "physical timestamps" for the event occurrences (e.g., event `C` occurs at 17ms521us). The discrete timeline shows how the same occurring events fit in the logical notion of time of Céu. The boot reaction `boot-0` happens before any input, at program startup. Event `A` "physically" occurs during `boot-0` but, because time is discrete, its corresponding reaction only executes afterwards, at logical instant `A-1`. Similarly, event `B` occurs during `A-1` and its reaction is postponed to execute at `B-2`. Event `C` also occurs during `A-1` but its reaction must also wait for `B-2` to execute and so it is postponed to execute at `C-3`. Finally, event `D` occurs during an idle period and can start immediately at `D-4`.

Unique input events imply mutually exclusive reactions, which execute atomically and never overlap. Automatic mutual exclusion is a prerequisite for deterministic reactions as we discuss in Section 3.

In practice, the synchronous hypothesis for Céu holds if reactions execute faster than the rate of incoming input events. Otherwise, the program would continuously accumulate delays between physical occurrences and actual reactions for the input events. Considering the context of soft real-time systems, postponed reactions might be tolerated as long as they are infrequent and the application does not take too long to catch up with real time. Note that the synchronous semantics is also the norm in typical event-driven systems, such as event dispatching in UI toolkits, game loops in game engines, and clock ticks in embedded systems.

2.1.2 Internal Events as Subroutines

In Céu, queue-based processing of events applies only to external input events, i.e., events submitted to the program by the environment. Internal events, which are events generated internally by the program via `emit` statements, are processed in a stack-based manner. Internal events provide a fine-grained execution control, and, because of their stack-based processing, can be used to implement a limited form of subroutines, as illustrated in Listing 3.

In the example, the "subroutine" `inc` is defined as an event iterator (ln 4–6) that continuously awaits its identifying event (ln 4), and increments the value passed by reference (ln 5). A trail in parallel (ln 8–11) invokes the subroutine through two consecutive `emit` statements (ln 9–10). Given the stack-based execution for internal events, as the first

emit executes, the calling trail pauses (ln 9), the subroutine awakes (ln 4), runs its body (yielding v=2), iterates, and awaits the next "call" (ln 4, again). Only after this sequence does the calling trail resumes (ln 9), makes a new invocation (ln 10), and passes the assertion test (ln 11).

```
1  event int* inc;          // declares subroutine "inc"
2  par/or do
3      var int* p;
4      every p in inc do // implements "inc" as event iterator
5          *p = *p + 1;
6      end
7  with
8      var int v = 1;
9      emit inc(&v);        // calls "inc"
10     emit inc(&v);        // calls "inc"
11     _assert(v==3);       // asserts result after the two returns
12 end
```

Listing 3. A "subroutine" that increments its argument.

Céu also supports nested `emit` invocations, e.g., the body of the subroutine `inc` (ln 5) could `emit` an event targeting another subroutine, creating a new level in the stack. We can think of the stack as a record of the nested, fine-grained internal reactions that happen inside the same outer reaction to a single external event.

This form of subroutines has a significant limitation that it cannot express recursion, since an `emit` to itself is always ignored as a running trail cannot be waiting on itself. That being said, it is this very limitation that brings important safety properties to subroutines. First, they are guaranteed to react in bounded time. Second, memory for locals is also bounded, not requiring data stacks.

At first sight, the constructions "`every e do ⟨···⟩ end`" and "`loop do await e; ⟨···⟩ end`" seem to be equivalent. However, the `loop` variation would not compile since it does not contain an external input `await` (`e` is an internal event). The `every` variation compiles because event iterators have an additional syntactic restriction that they cannot contain `break` or `await` statements. This restriction guarantees that an iterator never terminates from itself and, thus, always awaits its identifying event, essentially behaving as a safe blocking point in the program. For this reason, the restriction that execution paths within loops must contain at least one external `await` is extended to alternatively contain an `every` statement.

2.2 Shared-Memory Concurrency

Embedded applications make extensive use of global memory and shared resources, such as through memory-mapped registers and system calls to device drivers. Hence, an important goal of Céu is to ensure a reliable behavior for programs with concurrent lines of execution sharing memory and interacting with the environment.

In Céu, when multiple trails are active in the same reaction, they are scheduled in lexical order, i.e., in the order they appear in the program source code. For instance, consider the examples in Figure 2, both defining shared variables (ln 2), and assigning to them in parallel trails (ln 5,8).

In Listing 4, the two assignments to `x` can only execute in reactions to different events `A` and `B` (ln 4,7), which cannot occur simultaneously by definition. Hence, for the sequence of events `A->B`, `x` becomes `4` (`(1+1)*2`), while for `B->A`, `x` becomes `3` (`(1*2)+1`).

In Listing 5, the two assignments to `y` are simultaneous because they execute in reaction to the same event `A` (ln 4,7). Since Céu employs lexical order for intra-reaction statements, the execution is still deterministic, and `y` always becomes `4` (`(1+1)*2`). However, note that an apparently innocuous change in the order of trails modifies the behavior of the program. To mitigate this threat, Céu performs concurrency checks at compile time to detect conflicting accesses to shared variables: if a variable is written in a trail segment, then a concurrent trail segment cannot access that variable [20]. Nonetheless, the static checks are optional and are not a prerequisite for the deterministic semantics of the language.

```
input void A, B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
```

Listing 4.

```
1  input void A;
2  var int y = 1;
3  par/and do
4      await A;
5      y = y + 1;
6  with
7      await A;
8      y = y * 2;
9  end
```

Listing 5.

Figure 2. Shared-memory concurrency in Céu: Listing 4 is never concurrent because the trails access `x` atomically in different reactions; Listing 5 is concurrent, but still deterministic, because both trails access `y` atomically in the same reaction.

2.3 Abortion and Finalization

The `par/or` of Céu is an orthogonal abortion mechanism because the two sides in the composition need not be tweaked with synchronization primitives nor state variables to affect each other. In addition, abortion is *immediate* in the sense that it executes atomically inside the current micro reaction. Immediate orthogonal abortion is a distinctive feature of synchronous languages and cannot be expressed effectively in traditional (asynchronous) multi-threaded languages [4, 16].

However, aborting lines of execution that deal with resources may lead to inconsistencies. Therefore, Céu provides

a `finalize` construct to unconditionally execute a series of statements even if the enclosing block is externally aborted.

Céu also enforces, at compile time, the use of `finalize` for system calls that deal with pointers representing resources, as illustrated in the two examples of Figure 3. If Céu *passes* a pointer to a system call (Listing 6, ln 5), the pointer represents a *local* resource (ln 2) that requires finalization (ln 7). If Céu *receives* a pointer from a system call return (Listing 7, ln 4), the pointer represents an *external* resource (ln 2) that requires finalization (ln 6).

Céu tracks the interaction of system calls with pointers and requires finalization clauses to accompany them. In Listing 6, the local variable `msg` (ln 2) is an internal resource passed as a pointer to `_send` (ln 5), which is an asynchronous call that transmits the buffer in the background. If the block aborts (ln 11) before receiving an acknowledge from the environment (ln 9), the local `msg` goes out of scope and the external transmission now holds a *dangling pointer*. The finalization ensures that the transmission also aborts (ln 7). In Listing 7, the call to `_fopen` (ln 4) returns an external file resource as a pointer. If the block aborts (ln 12) during the `await A` (ln 9), the file remains open as a *memory leak*. The finalization ensures that the file closes properly (ln 6). In both cases, the code does not compile without the `finalize`.[2]

```
par/or do
   var _msg_t msg;
   ⟨···⟩    // prepare msg
   finalize
      _send(&msg);
   with
      _cancel(&msg);
   end
   await SEND_ACK;
with
   ⟨···⟩
end
```

Listing 6. Local resource.

```
1  par/or do
2     var _FILE* f;
3     finalize
4        f = _fopen(...);
5     with
6        _fclose(f);
7     end
8     _fwrite(..., f);
9     await A;
10    _fwrite(..., f);
11 with
12    ⟨···⟩
13 end
```

Listing 7. External resource.

Figure 3. Céu enforces the use of finalization to prevent dangling pointers and memory leaks.

The finalization mechanism of Céu is fundamental to preserve the orthogonality of the `par/or` construct since the clean up code is encapsulated in the aborted trail itself.

2.4 First-Class Timers

Embedded systems typically rely on activities that react to *wall-clock time*[3], such as timeout watchdogs and periodic sensor readings [8]. Céu supports wall-clock timers through the `await` statement [19], as illustrated in Listing 8. The `await` blocks the current running trail for the specified amount of time (ln 2,4).

However, underlying system timers do not activate programs immediately with zero delay due to intrinsic overhead, such as OS scheduling, interrupts with higher priorities, or even losses due to clock resolutions. We define the difference between a requested timeout and the actual expiring time as the *residual delta time (delta)*. Without explicit manipulation, the recurrent use of timed activities in a row (or in a loop) may accumulate a considerable amount of deltas that can lead to incorrect behavior in programs.

Céu handles deltas automatically, as illustrated in Listing 8. Suppose that after the first `await 10ms` request (ln 2), the underlying system gets busy and takes 15ms to notify Céu. The scheduler will notice that the `await` has not only already expired, but is delayed with `delta=5ms`. The awaiting trail awakes, sets `v=1` (ln 3), and then invokes `await 1ms` (ln 4). Notice that non-awaiting statements are considered to take no time in the synchronous model (e.g, ln 3). Since the current delta is still higher than the requested timeout (i.e. $5ms > 1ms$), the trail is rescheduled for execution, now with `delta=4ms`.

Delta compensation also makes timers in parallel to be perfectly synchronized. In the example in Listing 9, although the scheduler cannot guarantee that the first trail terminates exactly in 11ms (ln 2,4), it can at least ensure that it terminates before the second trail (yielding `v=1`), because $10 + 1 < 12$. A similar program in a language without first-class support for timers would depend on the execution timings for the code marked as <...> (ln 3), making the reasoning about the execution behavior more difficult.

```
var int v;
await 10ms;
v = 1;
await 1ms;
v = 2;
```

Listing 8. Wall-clock await.

```
1  par/or do
2    await 10ms;
3    <...> // non awaiting stmts
4    await  1ms;
5    v = 1;
6  with
7    await 12ms;
8    v = 2;
9  end
```

Listing 9. Synchronized timers.

Figure 4. First-class timers in Céu.

3 Formal Semantics of Céu

We now introduce and formalize the semantics of a reduced version of Céu, called *basic Céu*. Although simpler than the full language presented in Section 2, basic Céu is expressive

[2] The compiler only forces the programmer to write the finalization clause, but cannot check if it actually handles the resource properly.

[3] By wall-clock time we mean the passage of time from the real world, measured in hours, minutes, etc.

enough to capture all the essential characteristics of full Céu, in particular, the stack-based execution of internal events. Once basic Céu is defined, the more elaborate constructs of full Céu can be defined on top of it, as we will discuss shortly.

The statements of basic Céu are presented in Figure 5. In the figure, the metavariables v (ln 1, 2, 16), e (ln 3–5, 10), and n (ln 15–16) range over variable identifiers, event identifiers, and integers. The metavariable *stmt* (ln 1, 7–13, 16–19) denotes a statement, i.e., any of the statements of Figure 5—complex statements are defined recursively in terms of simpler statements. Finally, the metavariable *expr* (ln 2, 7) denotes an expression.

1	`var` v *stmt*	*variable declaration*
2	v := *expr*	*assignment statement*
3	`await`$_{ext}(e)$	*await external event*
4	`await`$_{int}(e)$	*await internal event*
5	`emit`$_{int}(e)$	*emit internal event*
6	`break`	*loop escape*
7	`if` *expr* `then` $stmt_1$ `else` $stmt_2$	*conditional*
8	$stmt_1$; $stmt_2$	*sequence*
9	`loop` *stmt*	*infinite loop*
10	`every` *e stmt*	*event iteration*
11	$stmt_1$ `par/and` $stmt_2$	*par/and statement*
12	$stmt_2$ `par/or` $stmt_2$	*par/or statement*
13	`fin` *stmt*	*finalization statement*
14	`@nop`	*dummy statement*
15	`@runat`(n)	*run at stack level n*
16	`@var` v *n stmt*	*expanded var*
17	$stmt_1$ `@loop` $stmt_2$	*expanded loop*
18	$stmt_1$ `@par/and` $stmt_2$	*expanded par/and*
19	$stmt_1$ `@par/or` $stmt_2$	*expanded par/or*

Figure 5. The statements of basic Céu.

For simplicity, we only consider integer expressions. These are build up from integer constants and variables by the usual mathematical operators (+, −, ≤, ...). We assume that expression evaluation takes zero time (in accordance with the synchronous hypothesis) and that it always produces an integer value. In places where a boolean value is expected, any nonzero value means true while zero means false.

We distinguish between three kinds of basic Céu statements. First, there are those statements which are common in imperative languages and behave as usual, namely, declarations, assignments, conditionals, sequences, loops, and breaks. (The `var` statement of basic Céu introduces a single local variable whose scope is the statement immediately following it.)

The statements of the second kind are those which are specific to Céu. These are the statements `await`$_{ext}$, `await`$_{int}$, `emit`$_{int}$, and `every`, which deal with events, the statements `par/and` and `par/or`, which define parallel compositions, and the statement `fin`, which defines a finalization block. These basic Céu statements are more or less equivalent to their counterparts in full Céu. We defer a precise description of their behavior and entailed properties to Section 3.3.

Finally, the statements of the third kind are the remaining ones, namely, `@nop`, `@runat`, `@var`, `@loop`, `@par/and`, and `@par/or`. These are hidden internal statements used by the interpreter to encode in the program's text information about its execution. We will have more to say about these `@`-statements in Section 3.3. Before that, however, we need to present the syntactical restrictions of basic Céu and discuss the mapping of full Céu programs into basic Céu programs.

3.1 Syntactic Restrictions of Basic Céu

The syntax of basic Céu shown in Figure 5 can be seen as a schema for generating programs. Not all programs generated by this schema are well-formed though. To be considered a *well-formed* basic Céu program, the generated program must satisfy the following restrictions:

1. If variable v occurs in an expression or assignment statement of the program, then this occurrence happens in the body of a `var` statement that declares v.
2. If a `break` occurs in the program, then this occurrence happens in the body of a `loop`.
3. If a statement of the form `loop` *stmt* occurs in the program, then all execution paths within *stmt* contain a matching `break` or an `await`$_{ext}$ or an `every`.
4. If a statement of the form `every` *e stmt* or `fin` *stmt* occurs in the program, then *stmt* does not contain occurrences of `loop`, `break`, `await`$_{ext}$, `await`$_{int}$, `every`, or `fin`.

Restrictions 1 and 2 prevent the use of undeclared variables and orphan `break`'s. Restriction 3 ensures that the program does not have an infinite loop with a body that runs in zero time (which would violate the synchronous hypothesis). And restriction 4 ensures that the body of `every` and `fin` statements always execute to completion within the same reaction. Similar restrictions exist in full Céu, as discussed in Section 2.

From now on, whenever we speak of a basic Céu program we mean a well-formed basic Céu program.

3.2 From Full Céu to Basic Céu

Most statements of full Céu are also present in basic Céu. These statements in common, however, are not exactly equivalent. It is sometimes the case that a statement of full Céu (say `await`) has more features than its basic Céu counterpart (`await`$_{int}$). In this section, we discuss how the extra features of full Céu are implemented in basic Céu.

Await and emit

The `await` and `emit` primitives of full Céu are slightly more complex than those of basic Céu, as they support communication of values between them.

Figure 6 shows a translation that adds a variable to hold the value being communicated. The original full Céu code in Listing 10 declares an internal event `e` (ln 2) and has an `await` (ln 4) and an `emit` (ln 10) that communicate the value 1 between the trails in parallel. The translation to basic Céu in Listing 11 declares an additional shared variable `e_` (ln 1) to hold the emitted value (ln 9) and which can be accessed by the awaking trail (ln 5).

External events require a similar translation, i.e., each event needs a corresponding global variable shared between all awaiting trails.

```
event int e;
par/or do
  var int v = await e;

  <...>
with
  <...>

  emit e(1);
end
```

Listing 10.

```
1  var int e_;
2  event int e;
3  par/or do
4    await e;
5    var int v = e_;
6    <...>
7  with
8    <...>
9    e_ = 1;
10   emit e;
11 end
```

Listing 11.

Figure 6. Full-to-Basic translation for `await` and `emit`.

First-class timers

To add support for first-class timers to basic Céu, we introduce a `TIMER` input event, which notifies the passage of time, and two global variables: `timer_`, which holds the elapsed time, and `delta_`, which holds the residual time (initially set to 0).

Listing 13 shows the basic Céu program resulting from the translation of the two timers shown in Listing 12 (ln 1–11). A timer first adds a (possibly) negative delta from the time it should await (ln 1). Then, it enters in a loop that awakes on every occurrence of `TIMER` (ln 7) and decrements the time it should await (ln 8). Each iteration of the loop checks if the timer has expired (ln 3), and sets the new delta, which may affect a timer in sequence. The check happens before the await because the timer may already start expired due to the residual time.

Finalization

The biggest mismatch between full Céu and basic Céu is in their support for finalization, i.e., between the statements `finalize` of full Céu and `fin` of basic Céu. Listing 14 shows a

```
await 10ms;
```

Listing 12.

```
1  var int tot_ = 10000 + delta_;
2  loop do
3      if tot_ <= 0 then
4          delta_ = tot_;
5          break;
6      end
7      await TIMER;
8      tot_ = tot_ - timer_;
9  end
10
11 var int tot_ = 1000 + delta_;
12 <...> // same loop as above
```

Listing 13.

Figure 7. Full-to-Basic translation for timers.

full Céu program containing an explicit block (ln 1–8) that executes the statements in `<A>` (ln 3) immediately followed by `<C>` (ln 7), and unconditionally executes `<B>` (ln 5) when the block terminates or aborts. To simulate this behavior in basic Céu we need to perform the translation shown in Listing 15. The basic Céu code also executes `<A>` (ln 1) immediately followed by `<C>` (ln 5). The difference is that the basic `fin <B>` statement (ln 3) blocks the pending statement forever, which only awakes and executes when it is aborted. The `par/or` (ln 2–6) serves this purpose since it allows `<C>` to execute immediately and aborts the `fin` when terminating.

```
do
    finalize
        <A>
    with
        <B>
    end
    <C>
end
```

Listing 14.

```
1  <A>
2  par/or do
3      fin <B>
4  with
5      <C>
6  end
7
8
```

Listing 15.

Figure 8. Full-to-Basic translation for finalization.

3.3 Operational Semantics

We proceed to formalize the operation of the basic Céu interpreter. Our goal here is twofold. First, we want to define a function *reaction* that describes precisely the operation steps taken by the interpreter to compute a single reaction to an external event. Second, we want to establish (prove) some properties of this function. In particular, we want to establish that:

1. *reaction* is indeed a function: the same program will always react in the same way to a same external event (i.e., reactions are deterministic);

2. *reaction* is a total function: its computation always yields a result (i.e., reactions terminate); and
3. *reaction* can be computed by a linear bounded automaton: the amount of memory it uses never exceeds a fixed threshold which depends solely on the input program (i.e., reactions are memory-bounded).

We will define *reaction* using a set of rules for a small-step operational semantics [17]. These rules dictate how the internal state of the basic Céu interpreter progresses while it is computing a reaction. A snapshot of this internal state is called a *description*, denoted δ, and consists of a quadruple $\langle stmt, \ell, e, \theta \rangle$ where

- *stmt* is a well-formed basic Céu program;
- ℓ is a nonnegative integer, called the stack level;
- e is an event identifier or the empty identifier ε; and
- θ is a memory.

We will detail the precise meaning and purpose of the components of the description in due course. For now, the important thing is that the steps taken by the interpreter to compute a reaction can be viewed as transitions between descriptions. The transitions are dictated by rules hardcoded in the interpreter. Each rule establishes that when the interpreter is in a description δ and certain criteria are met, then it will *transition* to a modified description δ', in symbols,

$$\delta \longrightarrow \delta'.$$

We call the description on the left-hand side of the symbol $\longrightarrow$ the *input description*, and the one on its right-hand side the *output description*.

After transitioning to a modified description, the interpreter repeats the rule-evaluation/transition process, and continues to do so until a final description is reached. This final description, called an *irreducible description*, embodies the result of the reaction.

A full *reaction* is thus defined as a sequence of the transitions of the form

$$\delta_0 \longrightarrow \delta_1 \longrightarrow \cdots \longrightarrow \delta_f.$$

The initial description $\delta_0 = \langle stmt_0, 0, e, \theta_0 \rangle$ contains the three inputs of the reaction: the text of the program at the beginning of the reaction ($stmt_0$), the event e to which the program must react, and the memory θ_0 holding the values of the variables of $stmt_0$. The final description $\delta_f = \langle stmt_f, 0, \varepsilon, \theta_f \rangle$ contains the two outputs of the reaction: the text of the program at the end of the reaction ($stmt_f$) and the memory θ_f holding the values of the variables of $stmt_f$. The output program $stmt_f$ and memory θ_f are used by the interpreter as inputs in the next reaction.

We write $\delta_0 \xrightarrow{i} \delta_f$ to indicate that δ_0 leads to δ_f after exactly i transitions, and we write $\delta_0 \xrightarrow{*} \delta_f$ to indicate that it does so after an unspecified but finite number of transitions. Using this notation, we can define function *reaction* (the first part of our goal) as follows:

$$reaction(stmt_0, \theta_0, e) = \langle stmt_f, \theta_f \rangle$$

if, and only if,

$$\langle stmt_0, 0, e, \theta_0 \rangle \xrightarrow{*} \langle stmt_f, 0, \varepsilon, \theta_f \rangle,$$

where $\langle stmt_f, 0, \varepsilon, \theta_f \rangle$ is an irreducible description. Under this definition, *reaction* will be deterministic, terminating, and memory-bounded (the second part of goal) if relation $\xrightarrow{*}$ happens to be so. That this is the case is a consequence of the way transitions are defined, as we will see in Section 4.

The next two sections, Sections 3.4 and 3.5, give the rules for transitions. There are two types of transitions: *outermost transitions* $\xrightarrow[out]{}$ and *nested transitions* $\xrightarrow[nst]{}$. The rules for $\xrightarrow[out]{}$ and $\xrightarrow[nst]{}$ transitions are listed all at once in Figure 9 (page 10). Each of these rules has the form

$$\frac{\text{condition}_1 \quad \text{condition}_2 \quad \cdots \quad \text{condition}_n}{\delta \longrightarrow \delta'}$$

and establishes that a transition $\delta \longrightarrow \delta'$ shall take place if condition_1, condition_2, …, and condition_n are all true. If the number of conditions is zero, then the line is omitted and the rule is called an axiom.

3.4 Outermost Transitions

The rules **push** and **pop** for $\xrightarrow[out]{}$ transitions (see Figure 9) are non-recursive definitions that apply to the program as a whole. These are the only rules that manipulate the stack level—the component of descriptions that determines the order of execution of nested reactions.

Rule **push** matches whenever there is a nonempty event in the input description; it instantly broadcasts the event to the program, which means it: (i) uses function *bcast* (defined below) to awake active $\texttt{await}_{ext}$, $\texttt{await}_{int}$, and `every` statements; (ii) creates a nested reaction by increasing the stack level (ℓ becomes $\ell + 1$); and (iii) consumes the input event (e becomes ε).

Function *bcast* is defined as

$$\begin{aligned}
bcast(\texttt{await}_{ext}(e), e) &= \texttt{@nop} \\
bcast(\texttt{await}_{int}(e), e) &= \texttt{@nop} \\
bcast(\texttt{every}\ e\ stmt, e) &= stmt; \texttt{every}\ e\ stmt \\
bcast(stmt_1; stmt_2, e) &= bcast(stmt_1, e); stmt_2 \\
bcast(\texttt{@var}\ v\ n\ stmt) &= \texttt{@var}\ v\ n\ \ bcast(stmt, e) \\
bcast(stmt_1\ \texttt{@loop}\ stmt_2, e) &= bcast(stmt_1, e)\ \texttt{@loop}\ stmt_2 \\
bcast(stmt_1\ \texttt{@par/and}\ stmt_2, e) &= \\
&\quad bcast(stmt_1, e)\ \texttt{@par/and}\ bcast(stmt_2, e) \\
bcast(stmt_1\ \texttt{@par/or}\ stmt_2, e) &= \\
&\quad bcast(stmt_1, e)\ \texttt{@par/or}\ bcast(stmt_2, e) \\
bcast(_, e) &= _ \quad \textit{(otherwise)}
\end{aligned}$$

where the underscore (_) denotes the omitted patterns.

Function *bcast* takes a statement *stmt* and an event e and awakes any trails in *stmt* awaiting for e. It does so by converting any active $\texttt{await}_{ext}(e)$ and $\texttt{await}_{int}(e)$ statements in *stmt*

to `@nop` (and thus consuming them) and by unwinding once any active `every` *e stmt′* statements in *stmt* (and thus allowing their body *stmt′* to execute once).

Note that **push** is the only rule that uses function *bcast*. Moreover, it is the only rule in the semantics that matches a nonempty event ($e \neq \varepsilon$) in the input description.

Rule **pop**, the other $\xrightarrow[out]{}$ rule, simply decreases the stack level by one. Rule **pop** can be applied whenever the stack level is greater than zero ($\ell > 0$) and the program is blocked or terminated (*stmt* = `@nop`). As we will see, these conditions ensure that an $\texttt{emit}_{int}$ only resumes after its nested reaction completes and blocks at the current stack level.

In order to determine whether the program is blocked, **pop** uses predicate *isblk*. This predicate evaluates to *true* if all parallel trails in the given program are blocked waiting for events, finalization clauses, or certain stack levels; otherwise, it evaluates to *false*. Predicate *isblk* is defined as

$$\begin{aligned}
isblk(\texttt{await}_{ext}(e), \ell) &= true \\
isblk(\texttt{await}_{int}(e), \ell) &= true \\
isblk(\texttt{every}\ e\ stmt, \ell) &= true \\
isblk(\texttt{@runat}(\ell'), \ell) &= (\ell > \ell') \\
isblk(\texttt{fin}\ stmt, \ell) &= true \\
isblk(stmt_1; stmt_2, \ell) &= isblk(stmt_1, \ell) \\
isblk(\texttt{@var}\ v\ \ell\ stmt, \ell) &= isblk(stmt, \ell) \\
isblk(stmt_1\ \texttt{@loop}\ stmt_2, \ell) &= isblk(stmt_1, \ell) \\
isblk(stmt_1\ \texttt{@par/and}\ stmt_2, \ell) &= \\
&\quad isblk(stmt_1, \ell) \wedge isblk(stmt_2, \ell) \\
isblk(stmt_1\ \texttt{@par/or}\ stmt_2, \ell) &= \\
&\quad isblk(stmt_1, \ell) \wedge isblk(stmt_2, \ell) \\
isblk(_, \ell) &= false \quad (otherwise)
\end{aligned}$$

where the underscore (_) denotes the omitted patterns.

Besides **pop**, predicate *isblk* is also used as a condition in the $\xrightarrow[nst]{}$ rules for advancing the right-hand side of parallel compositions (which can only advance if the left-hand side is blocked). We detail these and the other $\xrightarrow[nst]{}$ rules next.

3.5 Nested Transitions

The $\xrightarrow[nst]{}$ transitions have the general form

$$\langle stmt, \ell, \varepsilon, \theta \rangle \xrightarrow[nst]{} \langle stmt', \ell, e, \theta' \rangle.$$

They do not affect the stack level and never have an emitted event as a precondition. The distinction between $\xrightarrow[out]{}$ and $\xrightarrow[nst]{}$ prevents rules **push** and **pop** from matching and, consequently, from inadvertently modifying the stack level before a nested reaction is over.

A full reaction follows the pattern

$$\langle stmt_0, 0, e_{ext}, \theta_0 \rangle \xrightarrow[out]{push} \left[\xrightarrow[nst]{}^{*} \xrightarrow[out]{} \right] * \xrightarrow[nst]{}^{*} \xrightarrow[out]{pop} \langle stmt_f, 0, \varepsilon, \theta_f \rangle.$$

First, a $\xrightarrow[out]{push}$ starts a nested reaction at level 1. Then, a series of alternations between zero or more $\xrightarrow[nst]{}$ transitions and a single $\xrightarrow[out]{}$ transition (stack operation) takes place. Finally, zero or more $\xrightarrow[nst]{}$ transitions followed by a last $\xrightarrow[out]{pop}$ transition, which decrements the stack level to 0, terminates the reaction.

We now describe the rules for nested transitions (listed in Figure 9). Since the $\xrightarrow[nst]{}$ rules do not affect the stack level, whenever convenient, we omit the "ℓ" when discussing them. Thus, in Figure 9, we sometimes write descriptions as triples $\langle stmt, e, \theta \rangle$ with the tacit understanding that there is a hidden integer ℓ for the stack level between the components *stmt* and *e*.

Emissions

An $\texttt{emit}_{int}(e)$ generates event *e* and becomes a $\texttt{@runat}(\ell)$ statement (rule **emit-int**); this $\texttt{@runat}(\ell)$ statement only resumes at stack level ℓ (rule **run-at**).

Since every $\xrightarrow[nst]{}$ rule expects the input event to be empty, an application of **emit-int** leads immediately to an application of **push** at the outer level, creating a new level $\ell + 1$ on the stack. With this new stack level, the $\texttt{@runat}(\ell)$ resulting from the $\texttt{emit}_{int}(e)$ can only transition after future applications of **pop** put the stack level back in ℓ.

The interplay between **emit-int**, **push**, **pop**, and **run-at** provides the desired stack-based semantics for internal events.

Variable declarations and assignments

There are five $\xrightarrow[nst]{}$ rules for dealing with variables: **var-expd**, **var-nop**, **var-brk**, **var-adv**, and **assign**. The last two, **var-adv** and **assign**, are the only rules in the semantics that modify the memory—the fourth component of descriptions.

A *memory* θ is a stack of bindings $[b_1, \ldots, b_k]$ where each binding b_i is a pair (v, n) associating the variable v to the integer value n. For instance, the memory

$$[(v_1, 1), (v_2, \bot), (v_1, 0)]$$

has three bindings: $b_1 = (v_1, 1)$, which is the top-of-stack, $b_2 = (v_2, \bot)$, and $b_3 = (v_1, 0)$. From this memory, we can tell that, at some point, variable v_1 was declared and set to 0. Later, in the scope of v_1, variable v_2 was declared but not initialized, hence its value $\bot$ (undefined), and in the scope of v_2, variable v_1 was re-declared (shadowing the initial declaration) and set to 1.

The basic Céu interpreter starts with the empty memory $\theta = []$. When it encounters a declaration `var` *v stmt*, the interpreter immediately expands it to `@var` $v\ \bot$ *stmt* (rule **var-expd**) and proceeds to advance its body (*stmt*). At this point, three things can happen:

1. If *stmt* is terminated (= `@nop`), the interpreter consumes the whole `@var` statement by transforming it into a `@nop` (rule **var-nop**).
2. If *stmt* is a break (= `break`), the interpreter propagates the break outwards by transforming the whole `@var` statement into a `break` (rule **var-brk**).

Outermost transitions

push: $$\frac{e \neq \varepsilon}{\langle stmt, \ell, e, \theta\rangle \xrightarrow[out]{} \langle bcast(stmt, e), \ell + 1, \varepsilon, \theta\rangle}$$

pop: $$\frac{\ell > 0 \quad stmt = \texttt{@nop} \vee isblk(stmt, n)}{\langle stmt, \ell, \varepsilon, \theta\rangle \xrightarrow[out]{} \langle stmt, \ell - 1, \varepsilon, \theta\rangle}$$

Assignments & conditionals

assign: $\langle v := expr, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{@nop}, \varepsilon, updt(\theta, v, eval(\theta, expr))\rangle$

if-true: $$\frac{eval(\theta, expr) \neq 0}{\langle \texttt{if } expr \texttt{ then } stmt_1 \texttt{ else } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1, \varepsilon, \theta\rangle}$$

if-false: $$\frac{eval(\theta, expr) = 0}{\langle \texttt{if } expr \texttt{ then } stmt_1 \texttt{ else } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_2, \varepsilon, \theta\rangle}$$

Variable declarations

var-expd: $\langle \texttt{var } v\ stmt, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{@var } v \perp stmt, \varepsilon, \theta\rangle$

var-nop: $\langle \texttt{@var } v\ n\ \texttt{@nop}, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{@nop}, \varepsilon, \theta\rangle$

var-brk: $\langle \texttt{@var } v\ n\ \texttt{break}, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{break}, \varepsilon, \theta\rangle$

var-adv: $$\frac{\langle stmt, \varepsilon, (v, n){:}\theta\rangle \xrightarrow[nst]{} \langle stmt', e, (v, n'){:}\theta'\rangle}{\langle \texttt{@var } v\ n\ stmt, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{@var } v\ n'\ stmt', e, \theta'\rangle}$$

Par/and

par/and-expd:
$$\langle stmt_1 \texttt{ par/and } stmt_2, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1 \texttt{ @par/and } (\texttt{@runat}(\ell); stmt_2), \ell, \varepsilon, \theta\rangle$$

par/and-nop1:
$\langle \texttt{@nop @par/and } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_2, \varepsilon, \theta\rangle$

par/and-nop2:
$$\frac{isblk(stmt_1, \ell)}{\langle stmt_1 \texttt{ @par/and @nop}, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1, \ell, \varepsilon, \theta\rangle}$$

par/and-brk1:
$\langle \texttt{break @par/and } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle clear(stmt_2); \texttt{break}, \varepsilon, \theta\rangle$

par/and-brk2:
$$\frac{isblk(stmt_1, \ell)}{\langle stmt_1 \texttt{ @par/and break}, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle clear(stmt_1); \texttt{break}, \ell, \varepsilon, \theta\rangle}$$

par/and-adv1:
$$\frac{\langle stmt_1, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1', e, \theta'\rangle}{\langle stmt_1 \texttt{ @par/and } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1' \texttt{ @par/and } stmt_2, e, \theta'\rangle}$$

par/and-adv2:
$$\frac{isblk(stmt_1, \ell) \quad \langle stmt_2, \ell, \varepsilon, \theta'\rangle \xrightarrow[nst]{} \langle stmt_2', \ell, e, \theta'\rangle}{\langle stmt_1 \texttt{ @par/and } stmt_2, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1 \texttt{ @par/and } stmt_2', \ell, e, \theta'\rangle}$$

Emissions

emit-int: $\langle \texttt{emit}_{int}(e), \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{@runat}(\ell), \ell, e, \theta\rangle$

run-at: $$\frac{\ell' = \ell}{\langle \texttt{@runat}(\ell'), \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{@nop}, \ell, \varepsilon, \theta\rangle}$$

Sequences

seq-nop: $\langle \texttt{@nop}; stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_2, \varepsilon, \theta\rangle$

seq-brk: $\langle \texttt{break}; stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{break}, \varepsilon, \theta\rangle$

seq-adv: $$\frac{\langle stmt_1, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1', e, \theta'\rangle}{\langle stmt_1; stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1'; stmt_2, e, \theta'\rangle}$$

Loops

loop-expd: $\langle \texttt{loop } stmt, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt \texttt{ @loop } stmt, \varepsilon, \theta\rangle$

loop-nop: $\langle \texttt{@nop @loop } stmt, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{loop } stmt, \varepsilon, \theta\rangle$

loop-brk: $\langle \texttt{break @loop } stmt, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle \texttt{@nop}, \varepsilon, \theta\rangle$

loop-adv: $$\frac{\langle stmt_1, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1', e, \theta'\rangle}{\langle stmt_1 \texttt{ @loop } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1' \texttt{ @loop } stmt_2, e, \theta'\rangle}$$

Par/or

par/or-expd:
$$\langle stmt_1 \texttt{ par/or } stmt_2, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1 \texttt{ @par/or } (\texttt{@runat}(\ell); stmt_2), \ell, \varepsilon, \theta\rangle$$

par/or-nop1:
$\langle \texttt{@nop @par/or } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle clear(stmt_2), \varepsilon, \theta\rangle$

par/or-nop2:
$$\frac{isblk(stmt_1, \ell)}{\langle stmt_1 \texttt{ @par/or @nop}, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle clear(stmt_1), \ell, \varepsilon, \theta\rangle}$$

par/or-brk1:
$\langle \texttt{break @par/or } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle clear(stmt_2); \texttt{break}, \varepsilon, \theta\rangle$

par/or-brk2:
$$\frac{isblk(stmt_1, \ell)}{\langle stmt_1 \texttt{ @par/or break}, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle clear(stmt_1); \texttt{break}, \ell, \varepsilon, \theta\rangle}$$

par/or-adv1:
$$\frac{\langle stmt_1, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1', e, \theta'\rangle}{\langle stmt_1 \texttt{ @par/or } stmt_2, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1' \texttt{ @par/or } stmt_2, e, \theta'\rangle}$$

par/or-adv2:
$$\frac{isblk(stmt_1, \ell) \quad \langle stmt_2, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_2', \ell, e, \theta'\rangle}{\langle stmt_1 \texttt{ @par/or } stmt_2, \ell, \varepsilon, \theta\rangle \xrightarrow[nst]{} \langle stmt_1 \texttt{ @par/or } stmt_2', \ell, e, \theta'\rangle}$$

Figure 9. Rules for $\xrightarrow[out]{}$ and $\xrightarrow[nst]{}$ transitions.

3. Otherwise, the interpreter advances *stmt* recursively (rule **var-adv**) and updates the `@var` statement with the resulting body *stmt′*, event e, memory θ', and binding (v, n').

In rule **var-adv**, the recursive execution of the body (*stmt*) of the `@var` statement happens in a memory $(v, n) : \theta$, i.e., the memory resulting from pushing binding (v, n) to the current memory θ, which effectively declares v with value n in the scope of *stmt*. Initially, $n = \perp$ (undefined), but any assignments in *stmt* may change that. Thus, after *stmt* is advanced, the value n' associated with variable v in the (possibly) updated binding (v, n') is stored back in the enclosing `@var` statement. (The updated binding itself is popped from the resulting memory because at this point the `@var` statement is no longer in effect.)

We borrowed the technique used in rule **var-adv** to implement lexical scoping from the operational semantics of Esterel [7]. Our version is simpler though. While in Esterel the memories resulting from advancing parallel trails need to be merged into a single memory, in Céu this is not necessary: parallel trails are always advanced from left to right and can thus safely operate over the same memory.

The last $\xrightarrow[nst]{}$ rule for dealing with variables is **assign**. Rule **assign** transforms $v := expr$ into `@nop` and performs the assignment. That is, it evaluates *expr* to an integer n and updates the most recent binding of variable v to n.

Expression evaluation is carried out by function *eval*. For simplicity, we assume that *eval* always returns an integer. In practice, any error in the evaluation, such as access to an uninitialized variable or division by zero, will cause the interpreter to abort.

Memory updates are carried out by function *updt*, which simply updates the most recent binding of the given variable to the given value:

$$updt((v, n) : \theta, v', n') = \begin{cases} (v, n') : \theta & \text{if } v = v' \\ (v, n) : updt(\theta, v', n') & \text{otherwise.} \end{cases}$$

The first syntactical restriction listed in Section 3.1 ensures that *updt* is always defined.

Conditionals

Rules **if-true** and **if-false** transform a conditional statement into its if-part ($stmt_1$) or else-part ($stmt_2$) depending on the value to which the associated expression (*expr*) evaluates in the current memory. If it evaluates to a nonzero value, then **if-true** applies and the conditional becomes $stmt_1$; otherwise, **if-false** applies and the conditional becomes $stmt_2$.

These are the only rules that use the contents of the memory in a way that affects the control flow.

Sequences

A sequence whose first part is terminated (`@nop`) becomes its second part (rule **seq-nop**). A sequence whose first part is a `break`, propagates the `break` outwards by dropping its second part (rule **seq-brk**). And a sequence whose first part is neither `@nop` nor `break` advances to the sequence, event, and memory resulting from advancing just its first part (rule **seq-adv**).

Loops

When the interpreter encounters a `loop` *stmt*, it unwinds it once, expanding it to *stmt* `@loop` *stmt* (rule **loop-expd**).

The remaining rules for loops are similar to those for sequences but use "`@`" as separators to bind the current code of the loop to its next iteration. Rules **loop-nop** and **loop-adv** are analogous to rules **seq-nop** and **seq-adv**. They advance the current code of the loop until it becomes a `@nop`. When this happens the loop is restarted (**loop-nop**). (Note that if we had used "`;`" as a separator in loops, rules **loop-brk** and **seq-brk** would conflict.) Finally, rule **loop-brk** escapes the enclosing loop, transforming everything into a `@nop`.

Par/and and par/or compositions

When the interpreter encounters `par/and` and `par/or`, it expands them to `@par/and` and `@par/or` prefixing a `@runat`(ℓ) to the right-hand side of the resulting compositions (rules **par/and-expd** and **par/or-expd**). The `@runat`(ℓ) ensures that any reachable $\texttt{emit}_{int}$ on the left-hand side executes to completion before the right-hand side starts.

There are twelve rules for advancing the sides of `@par/and` and `@par/or` statements. Each of these rules ensures that the left-hand side always advances before the right-hand side. That is, in each rule the right-hand side is advanced only when the left-hand side is blocked. The twelve rules are divided into three groups: (i) those that apply when one of the sides is `@nop`; (ii) those that apply when one of the sides is `break`; and (iii) those that apply when neither side is `@nop` or `break`.

We begin by presenting the rules of the last group, which do not involve the termination of one of the sides.

Rules **par/and-adv1** and **par/or-adv1** match whenever the left-hand side of the composition is not blocked (see *isblk* in Section 3.4). They advance the composition the to the statement, event, and memory resulting from advancing just its left-hand side. Rules **par/and-adv2** and **par/or-adv2** are similar, but they act on the right-hand side and match only when the left-hand side is blocked.

The next rules apply whenever one of the sides of the `@par/and` or `@par/or` terminates (becomes a `@nop`).

In a `@par/and`, if one of the sides terminates, the composition becomes the other side (rules **par/and-nop1** and **par/and-nop2**). In a `@par/or`, however, if one of the sides terminates, the whole composition terminates and function *clear* is called to finalize the aborted side (rules **par/or-nop1** and **par/or-nop2**).

Function *clear* takes a statement *stmt* and extracts the bodies of all active `fin` statements in *stmt*, returning them

in a sequence:

$$
\begin{aligned}
\mathit{clear}(\texttt{await}_{ext}(e)) &= \texttt{@nop} \\
\mathit{clear}(\texttt{await}_{int}(e)) &= \texttt{@nop} \\
\mathit{clear}(\texttt{every}\ e\ \mathit{stmt}) &= \texttt{@nop} \\
\mathit{clear}(\texttt{@runat}(\ell)) &= \texttt{@nop} \\
\mathit{clear}(\texttt{fin}\ \mathit{stmt}) &= \mathit{stmt} \\
\mathit{clear}(\mathit{stmt}_1; \mathit{stmt}_2) &= \mathit{clear}(\mathit{stmt}_1) \\
\mathit{clear}(\texttt{@var}\ v\ n\ \mathit{stmt}) &= \mathit{clear}(\mathit{stmt}) \\
\mathit{clear}(\mathit{stmt}_1\ \texttt{@loop}\ \mathit{stmt}_2) &= \mathit{clear}(\mathit{stmt}_1) \\
\mathit{clear}(\mathit{stmt}_1\ \texttt{@par/and}\ \mathit{stmt}_2) &= \mathit{clear}(\mathit{stmt}_1); \mathit{clear}(\mathit{stmt}_2) \\
\mathit{clear}(\mathit{stmt}_1\ \texttt{@par/or}\ \mathit{stmt}_2) &= \mathit{clear}(\mathit{stmt}_1); \mathit{clear}(\mathit{stmt}_2) \\
\mathit{clear}(\_) &= \bot \quad \textit{(undefined)}
\end{aligned}
$$

When applied to the aborted side of a `@par/or`, *clear* returns all finalization code associated with it (if any). Thus, by expanding to a *clear* call on the aborted side, rules **par/or-nop1** and **par/or-nop2**, ensure that the composition is properly finalized.

Note that as there are no transition rules for `fin` statements, once reached, a `fin` halts and will only be consumed if its trail is aborted. At this moment, its body will execute as a result of the *clear* call. Note also that the syntactic restrictions of Section 3.1 ensure that the body of a `fin` statement always execute within the same reaction.

Finally, a `break` in one of the sides of a `@par/and` or `@par/or` is propagated outwards, terminating the composition and properly finalizing the aborted side with a *clear* call (rules **par/and-brk1**, **par/and-brk2**, **par/or-brk1**, and **par/or-brk2**).

3.6 Using the Rules to Compute Reactions

We conclude the discussion of the rules for $\xrightarrow[out]{}$ and $\xrightarrow[nst]{}$ transitions with an example of their application in the computation of reactions.

Consider the the following basic Céu program:

```
var led
  (await_ext(e_bt)) par/or (loop (led := 1; await_ext(e_rd);
                                  led := 0; await_ext(e_rd));
                            fin led := 0)
```

This is a simplified basic Céu version of the full Céu program presented in Listing 2 (page 2). As its full Céu counterpart, this program toggles a LED whenever a radio packet (event e_{rd}) is received, terminating on a button press (event e_{bt}) always with the LED off—the state of the led is represented by variable *led*.

We will compute three reactions of the above program. First, to the bootstrap event e_0 (the bootstrap reaction) and then to the external events e_{rd} and e_{bt} (in this order). We will execute the reactions in series, which means that the outputs of a reaction will become inputs of the next reaction.

The bootstrap reaction starts with a description

$$\langle \texttt{var}\ \mathit{led}\ \ldots, 0, e_0, [] \rangle,$$

where `var` *led* … is the original program, e_0 is the bootstrap event, and [] is the empty memory, and proceeds as follows:[4]

$\xrightarrow{\text{push}}$ `var` *led* … $\quad \ell = 1, e = \varepsilon, \theta = []$

$\xrightarrow[\text{expd}]{\text{var}}$ `@var` *led* $\bot$ … `par/or` …

$\hookrightarrow$ … `par/or` … $\quad \theta = [(\mathit{led}, \bot)]$

$\xrightarrow[\text{expd}]{\text{par/or}}$ $\texttt{await}_{ext}(e_{bt})$ `@par/or` (`loop` …)

$\hookrightarrow$ `loop` …

$\xrightarrow[\text{expd}]{\text{loop}}$ *led* := 1; … `@loop` …

$\hookrightarrow$ *led* := 1; $\texttt{await}_{ext}(e_{rd})$; …

$\hookrightarrow$ *led* := 1

$\xrightarrow{\text{assign}}$ `@nop` $\quad \theta = [(\mathit{led}, 1)]$

$\xrightarrow[\text{nop}]{\text{seq}}$ $\texttt{await}_{ext}(e_{rd})$; …

$\xrightarrow[\text{adv}]{\text{loop}}$ $\texttt{await}_{ext}(e_{rd})$; … `@loop` …

$\xrightarrow[\text{adv2}]{\text{par/or}}$ $\texttt{await}_{ext}(e_{bt})$ `@par/or` ($\texttt{await}_{ext}(e_{rd})$; … `@loop` …)

$\xrightarrow[\text{adv}]{\text{var}}$ `@var` *led* 1 … `@par/or` … $\quad \theta = []$

$\xrightarrow{\text{pop}}$ `@var` *led* 1 … `@par/or` … $\quad \ell = 0$

So, the bootstrap reaction terminates with variable *led* set to 1 and with both sides of the `@par/or` blocked on $\texttt{await}_{ext}$ statements: the left-hand side waiting on e_{bt} and the right hand-side waiting on e_{rd}.

This is a general property of reactions. Every reaction eventually blocks in $\texttt{await}_{ext}$, $\texttt{await}_{int}$, `every`, `fin`, or `@runat` statements in parallel trails. If none of these trails is blocked in a `@runat`, then the program can no longer advance and the reaction is terminated.

The second reaction is a reaction to event e_{rd}. It starts with the description produced by the previous reaction but with ε replaced by e_{rd}, i.e.,

$$\langle \texttt{@var}\ \mathit{led}\ 1\ \texttt{await}_{ext}(e_{bt})\ \texttt{@par/or}(\texttt{await}_{ext}(e_{rd}); \mathit{led} := 0; \ldots \texttt{@loop} \ldots), 0, e_{rd}, [] \rangle$$

and after three transitions in the outer level (**push**, **var-adv**, and **pop**) results in:

$$\langle \texttt{@var}\ \mathit{led}\ 0\ \texttt{await}_{ext}(e_{bt})\ \texttt{@par/or}(\texttt{await}_{ext}(e_{rd}); \mathit{led} := 1; \ldots \texttt{@loop} \ldots), 0, \varepsilon, [] \rangle$$

where *led* is set to 0 and the event is consumed.

The third and last reaction is a reaction to event e_{bt}. It starts with the description produced in the previous reaction (with e_{bt} replaced for ε) and proceeds as follows:

[4] The arrow ($\rightarrow$) denotes an $\xrightarrow[out]{}$ or $\xrightarrow[nst]{}$ transition and the hook ($\hookrightarrow$) introduces a recursive $\xrightarrow[nst]{}$ transition. We only show the affected part of statements, and any updates on the level ℓ, event e, or memory θ appear on the right-hand side.

$$\xrightarrow{\text{push}} \texttt{@var}\ \mathit{led}\ 0\ \texttt{@nop}\ \texttt{@par/or}\ldots \qquad \ell = 1, e = \varepsilon, \theta = []$$
$$\hookrightarrow \texttt{@nop}\ \texttt{@par/or}\ldots \qquad \theta = [(\mathit{led}, 0)]$$
$$\xrightarrow[\text{nop1}]{\text{par/or}} \mathit{led} := 0 \qquad (= \mathit{clear}(\ldots))$$
$$\xrightarrow{\text{assign}} \texttt{@nop} \qquad \theta = [(\mathit{led}, 0)]$$
$$\xrightarrow[\text{adv}]{\text{var}} \texttt{@var}\ \mathit{led}\ 0\ \texttt{@nop} \qquad \theta = []$$
$$\xrightarrow[\text{nop}]{\text{var}} \texttt{@nop}$$
$$\xrightarrow{\text{pop}} \texttt{@nop} \qquad \ell = 0$$

After the third reaction the program is terminated (consists of a single `@nop`) and no longer responds to the environment.

4 Properties of Céu Reactions

In this section, we state and prove three properties of Céu reactions. First, that reactions are deterministic (Section 4.1). Second, that they always produce a result and terminate (Section 4.2). And third, that the maximum amount of memory they consume never exceeds a fixed threshold which depends solely on their inputs (Section 4.3).

These three properties are actually properties of transitions and are inherited by reactions. Recall that reactions are formally defined in terms of transitions as follows:

$$\mathit{reaction}(\mathit{stmt}_0, \theta_0, e) = \langle \mathit{stmt}_f, \theta_f \rangle$$
$$\text{iff} \quad \langle \mathit{stmt}_0, 0, e, \theta_0 \rangle \xrightarrow{*} \langle \mathit{stmt}_f, 0, \varepsilon, \theta_f \rangle .$$

In words, a basic Céu program stmt_0 in a memory θ_0 when fed with the external event e will react to a modified program stmt_f and memory θ_f if (and only if) a finite (possibly zero) number of transitions transforms the description $\langle \mathit{stmt}_0, 0, e, \theta_0 \rangle$ into $\langle \mathit{stmt}_f, 0, \varepsilon, \theta_f \rangle$.

So, from this definition, *reaction* will be deterministic, terminating, and memory bounded, if relation $\xrightarrow{*}$ is so.

4.1 Deterministic Execution

Transitions $\xrightarrow[out]{}$ and $\xrightarrow[nst]{}$ are defined in such a way that given an input description either no rule is applicable or exactly one of them can be applied (no choice involved). This coupled with the fact that the output of every rule is a function of its input implies that transitions are deterministic: the same input description, if it can transition, will always result in the same output description. Thus the transition relation $\longrightarrow$ is in fact a partial function.

The next two lemmas establish the determinism of a single application of $\xrightarrow[out]{}$ and $\xrightarrow[nst]{}$. Lemma 4.1 follows from a simple inspection of rules **push** and **pop**. The proof of Lemma 4.2 follows by induction on the structure of the derivation trees produced by the rules for $\xrightarrow[nst]{}$. Both lemmas are used in the proof of the Theorem 4.3.

Lemma 4.1. *If* $\delta \xrightarrow[out]{} \delta_1$ *and* $\delta \xrightarrow[out]{} \delta_2$ *then* $\delta_1 = \delta_2$.

Lemma 4.2. *If* $\delta \xrightarrow[nst]{} \delta_1$ *and* $\delta \xrightarrow[nst]{} \delta_2$ *then* $\delta_1 = \delta_2$.

The main result of this section, Theorem 4.3, establishes that any given number $i \geq 0$ of applications of arbitrary transition rules, starting from the same input description, will always lead to the same output description. In other words, any finite sequence of transitions behave deterministically.

Theorem 4.3 (Determinism). $\delta \xrightarrow{i} \delta_1$ *and* $\delta \xrightarrow{i} \delta_2$ *implies* $\delta_1 = \delta_2$.

Proof. By induction on i. The theorem is trivially true if $i = 0$ and follows directly from the previous lemmas if $i = 1$. Suppose

$$\delta \xrightarrow{1} \delta_1' \xrightarrow{i-1} \delta_1 \quad \text{and} \quad \delta \xrightarrow{1} \delta_2' \xrightarrow{i-1} \delta_2 ,$$

for some $i > 1$, δ_1' and δ_2'. Then, by Lemma 4.1 or 4.2, depending on whether the first transition is $\xrightarrow[out]{}$ or $\xrightarrow[nst]{}$ (it cannot be both), $\delta_1' = \delta_2'$, and by the induction hypothesis, $\delta_1 = \delta_2$. □

4.2 Terminating Reactions

We now turn to the problem of termination. We want to show that any sufficiently long sequence of applications of arbitrary transition rules will eventually lead to an irreducible description, i.e., one that cannot be modified by further transitions. Before doing that, however, we need to introduce some notation and establish some basic properties of the transition relations $\xrightarrow[nst]{}$ and $\xrightarrow[out]{}$.

Definition 4.4. A description $\delta = \langle p, n, e \rangle$ is *nested-irreducible* iff $e \neq \varepsilon$ or $p = \texttt{@nop}$ or $p = \texttt{break}$ or $\mathit{isblocked}(p, n)$.[5]

Nested-irreducible descriptions serve as normal forms for $\xrightarrow[nst]{}$ transitions: they embody the result of an exhaustive number of $\xrightarrow[nst]{}$ applications. We will write $\delta_{\#nst}$ to indicate that description δ is nested-irreducible.

The use of qualifier "irreducible" in Definition 4.4 is justified by Proposition 4.5, which states that if a finite number of applications of $\xrightarrow[nst]{}$ results in an irreducible description, then that occurs exactly once, at some specific number i. The proof of Proposition 4.5 follows directly from the definition of $\xrightarrow[nst]{}$ by contradiction on the hypothesis that there is such $k \neq i$.

Proposition 4.5. *If* $\delta \xrightarrow[nst]{i} \delta'_{\#nst}$ *then, for all* $k \neq i$, *there is no* $\delta''_{\#nst}$ *such that* $\delta \xrightarrow[nst]{k} \delta''_{\#nst}$.

The next lemma establishes that sequences of $\xrightarrow[nst]{}$ transitions behave as expected regarding the order of evaluation of composition branches. Its proof follows by induction on i.

Lemma 4.6.
If $\langle p_1, n, e \rangle \xrightarrow[nst]{i} \langle p_1', n, e' \rangle$, *for any* p_2:

(a) $\langle p_1; p_2, n, e \rangle \xrightarrow[nst]{i} \langle p_1'; p_2, n, e' \rangle$.
(b) $\langle p_1\ \texttt{@loop}\ p_2, n, e \rangle \xrightarrow[nst]{i} \langle p_1'\ \texttt{@loop}\ p_2, n, e' \rangle$.
(c) $\langle p_1\ \texttt{@and}\ p_2, n, e \rangle \xrightarrow[nst]{i} \langle p_1'\ \texttt{@and}\ p_2, n, e' \rangle$.

[5] We sometimes abbreviate "$p = \texttt{@nop}$ or $p = \texttt{break}$" as "$p = \texttt{@nop, break}$".

(d) $\langle p_1 \texttt{ @or } p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_1' \texttt{ @or } p_2, n, e'\rangle$.

If $\langle p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_2', n, e'\rangle$, *for any* p_1 *such that* $isblocked(p_1, n)$:

(a) $\langle p_1 \texttt{ @and } p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_1 \texttt{ @and } p_2', n, e'\rangle$.
(b) $\langle p_1 \texttt{ @or } p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_1 \texttt{ @or } p_2', n, e'\rangle$.

The syntactic restriction discussed in Section 2.1 regarding the body of loops and the restriction mentioned in Section 3.3 about the body of `fin` statements are formalized in Assumption 4.7 below. These restrictions are essential to prove the next theorem.

Assumption 4.7 (Syntactic restrictions)**.**

(a) If $p = \texttt{fin}\, p_1$ then p_1 contains no occurrences of statements $\texttt{await}_{ext}$, $\texttt{await}_{int}$, `every`, or `fin`. And so, for any n, $\langle clear(p_1), n, \varepsilon\rangle \xrightarrow[nst]{*} \langle \texttt{@nop}, n, \varepsilon\rangle$.
(b) If $p = \texttt{loop}\, p_1$ then all execution paths of p_1 contain a matching `break` or an $\texttt{await}_{ext}$. Consequently, for all n, there are p_1' and e such that $\langle \texttt{loop}\, p_1, n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p_1', n, e\rangle$, where $p_1' = \texttt{break @loop}\, p_1$ or $isblocked(p_1', n)$.

Theorem 4.8 establishes that a finite (possibly zero) number of $\xrightarrow[nst]{}$ transitions eventually leads to a nested-irreducible description. Hence, for any input description δ, it is always possible to transform δ in a nested-irreducible description δ' by applying to it a sufficiently long sequence of $\xrightarrow[nst]{}$ transitions. The proof of the theorem follows by induction on the structure of programs (members of set P) and depends on Lemma 4.6 and Assumption 4.7.

Theorem 4.8. *For any* δ *there is a* $\delta'_{\#nst}$ *such that* $\delta \xrightarrow[nst]{*} \delta'_{\#nst}$.

The main result of this section, Theorem 4.15, is similar to Theorem 4.8 but applies to transitions $\longrightarrow$ in general. Before stating and proving it, we need to characterize irreducible descriptions in general. This characterization, given in Definition 4.11, depends on the notions of potency and rank.

Definition 4.9. The *potency* of a program p in reaction to event e, denoted $pot(p, e)$, is the maximum number of $\texttt{emit}_{int}$ statements that can be executed in a reaction of p to e, i.e.,

$$pot(p, e) = pot'(bcast(p, e)),$$

where pot' is an auxiliary function that counts the maximum number of reachable $\texttt{emit}_{int}$ statements in the program resulting from the broadcast of event e to p.

Function pot' is defined by the following clauses:

(a) $pot'(\texttt{emit}_{int}(e)) = 1$.
(b) $pot'(\texttt{if mem}(id)\texttt{ then } p_1 \texttt{ else } p_2) = \max\{pot'(p_1), pot'(p_2)\}$.
(c) $pot'(\texttt{loop}\, p_1) = pot'(p_1)$.
(d) $pot'(p_1 \texttt{ and } p_2) = pot'(p_1 \texttt{ or } p_2) = pot'(p_1) + pot'(p_2)$.
(e) If $p_1 \neq \texttt{break}, \texttt{await}_{ext}(e)$,

$$pot'(p_1; p_2) = pot'(p_1) + pot'(p_2)$$

$$pot'(p_1 \texttt{ @loop } p_2) = \begin{cases} pot'(p_1) & \text{if } (\dagger) \\ pot'(p_1) + pot'(p_2) & \text{otherwise,} \end{cases}$$

where (†) stands for: "a `break` or $\texttt{await}_{ext}$ occurs in all execution paths of p_1".
(f) If $p_1, p_2 \neq \texttt{break}$, $pot'(p_1 \texttt{ @and } p_2) = pot'(p_1) + pot'(p_2)$.
(g) If $p_1, p_2 \neq \texttt{break}$ and $p_1, p_2 \neq \texttt{@nop}$,

$$pot'(p_1 \texttt{ @or } p_2) = pot'(p_1) + pot'(p_2).$$

(h) Otherwise, if none of (a)–(g) applies, $pot'(\_) = 0$.

Definition 4.10. The *rank* of a description $\delta = \langle p, n, e\rangle$, denoted $rank(\delta)$, is a pair of nonnegative integers $\langle i, j\rangle$ such that

$$i = pot(p, e) \quad \text{and} \quad j = \begin{cases} n & \text{if } e = \varepsilon \\ n + 1 & \text{otherwise.} \end{cases}$$

Intuitively, the rank of a description δ is a measure of the maximum amount of "work" (transitions) required to transform δ into an irreducible description, in the following sense.

Definition 4.11. A description δ is *irreducible* (in symbols, $\delta_\#$) iff it is nested-irreducible and its $rank(\delta)$ is $\langle i, 0\rangle$, for some $i \geq 0$.

An irreducible description $\delta_\# = \langle p, n, e\rangle$ serves as a normal form for transitions $\longrightarrow$ in general. Such description cannot be advanced by $\xrightarrow[nst]{}$, as it is nested-irreducible, and neither by $\xrightarrow[out]{push}$ nor $\xrightarrow[out]{pop}$, as the second coordinate of its rank is 0, which implies $e = \varepsilon$ and $n = 0$.

The next two lemmas establish that a single application of $\xrightarrow[out]{}$ or $\xrightarrow[nst]{}$ either preserves or decreases the rank of the input description. All rank comparisons assume lexicographic order, i.e., if $rank(\delta) = \langle i, j\rangle$ and $rank(\delta') = \langle i', j'\rangle$ then $rank(\delta) > rank(\delta')$ iff $i > i'$ or $i = i$ and $j > j'$. The proof of Lemma 4.12 follows directly from **push** and **pop** and from Definitions 4.9 and 4.10. The proof of Lemma 4.13, however, is by induction on the structure of $\xrightarrow[nst]{}$ derivations.

Lemma 4.12.

(a) *If* $\delta \xrightarrow[out]{push} \delta'$ *then* $rank(\delta) = rank(\delta')$.
(b) *If* $\delta \xrightarrow[out]{pop} \delta'$ *then* $rank(\delta) > rank(\delta')$.

Lemma 4.13. *If* $\delta \xrightarrow[nst]{} \delta'$ *then* $rank(\delta) \geq rank(\delta')$.

The next theorem is a generalization of Lemma 4.13 for $\xrightarrow[nst]{*}$. Its proof follows from the lemma by induction on i.

Theorem 4.14. *If* $\delta \xrightarrow[nst]{*} \delta'$ *then* $rank(\delta) \geq rank(\delta')$.

We now state and prove the main result of this section, Theorem 4.15, the termination theorem for $\xrightarrow{*}$. The idea of the proof is that a sufficiently large sequence of $\xrightarrow[nst]{}$ and $\xrightarrow[out]{}$ transitions eventually decreases the rank of the current description until an irreducible description is reached. This irreducible description is the final result of the reaction.

Theorem 4.15 (Termination)**.** *For any* δ, *there is a* $\delta'_\#$ *such that* $\delta \xrightarrow{*} \delta'_\#$.

Proof. By lexicographic induction on $rank(\delta)$. Let $\delta = \langle p, n, e \rangle$ and $rank(\delta) = \langle i, j \rangle$.

For the basis, suppose $\langle i, j \rangle = \langle 0, 0 \rangle$. Then δ cannot be advanced by $\xrightarrow[out]{}$, as $j = 0$ implies $e = \varepsilon$ and $n = 0$. If δ is nested-irreducible, the theorem is trivially true, as $\delta \xrightarrow[nst]{0} \delta_{\#nst}$ and $\delta_{\#}$. If δ is not nested-irreducible then, by Theorem 4.8, $\delta \xrightarrow[nst]{*} \delta'_{\#nst}$, for some $\delta'_{\#nst}$. By Theorem 4.14, $rank(\delta) \geq rank(\delta')$, which implies $rank(\delta') = \langle 0, 0 \rangle$, and so $\delta'_{\#}$.

For the inductive step, suppose $\langle i, j \rangle > \langle 0, 0 \rangle$. Then, depending on whether or not δ is nested-irreducible, there are two cases.

Case 1. δ is nested-irreducible. If $j = 0$, by Definition 4.11, $\delta_{\#}$, and so $\delta \xrightarrow{0} \delta_{\#}$. If $j > 0$, there are two subcases:

Case 1.1. $e \neq \varepsilon$. Then, by **push** and by Theorem 4.8, there are δ'_1 and $\delta'_{\#nst} = \langle p', n+1, e' \rangle$ such that $\delta \xrightarrow[out]{push} \delta'_1 \xrightarrow[nst]{*} \delta'_{\#nst}$. Thus, by Lemma 4.12 and by Theorem 4.14,

$$rank(\delta) = rank(\delta'_1) = \langle i, j \rangle \geq rank(\delta') = \langle i', j' \rangle .$$

If $e' = \varepsilon$, then $i = i'$ and $j = j'$, and the rest of this proof is similar to that of Case 1.2 below. Otherwise, if $e' \neq \varepsilon$, then $i > i'$, as an $\texttt{emit}_{int}(e')$ was consumed by the nested transitions. Thus, $rank(\delta) > rank(\delta')$. By the induction hypothesis, $\delta' \xrightarrow{*} \delta''_{\#}$, for some $\delta''_{\#}$. Therefore, $\delta \xrightarrow{*} \delta''_{\#}$.

Case 1.2. $e = \varepsilon$. Then, as $j > 0$, $\delta \xrightarrow[out]{pop} \delta'$, for some δ'. By Lemma 4.12, $rank(\delta) > rank(\delta')$. Hence, by the induction hypothesis, $\delta' \xrightarrow{*} \delta''_{\#}$, for some $\delta''_{\#}$. And so, $\delta \xrightarrow{*} \delta''_{\#}$.

Case 2. δ is not nested-irreducible. Then $e = \varepsilon$ and, by Theorems 4.8 and 4.14, there is a $\delta'_{\#nst}$ such that $\delta \xrightarrow[nst]{*} \delta'_{\#nst}$ with $rank(\delta) \geq rank(\delta'_{\#nst})$. The rest of this proof is similar to that of Case 1 above. □

4.3 Memory Boundedness

As Céu has no mechanism for heap allocation, unbounded iteration, or general recursion, the maximum memory usage of a given Céu program is determined solely by the length of its code, the number of variables it uses, and the size of the event stack that it requires to run. The code length and the number of variables used are easily determined by code inspection. The maximum size of the event stack during a reaction of program p to external event e corresponds to $pot(p, e)$, i.e., to the maximum number of internal events that p may emit in reaction to e. If p may react to external events $e_1, \ldots, e_n$ then, in the worst case, its event stack will need to store $\max\{pot(p, e_1), \ldots, pot(p, e_n)\}$ events.

5 Related Work

Céu follows the lineage of imperative synchronous languages initiated by Esterel [10]. These languages typically define time as a discrete sequence of logical "ticks" in which multiple simultaneous input events can be active [19]. The presence of multiple inputs requires careful static analysis to detect and reject programs with *causality cycles* and *schizophrenia problems* [5]. In contrast, Céu defines time as a discrete sequence of reactions to unique input events, which is a prerequisite for the concurrency checks that enable safe shared-memory concurrency, as discussed in Section 2.2.

In most synchronous languages, the behavior of external and internal events is equivalent. However, in Céu, internal events introduce stack-based micro reactions within external reactions, providing more fine-grained control for intra-reaction execution. This allows for memory-bounded subroutines that can execute multiple times during the same external reaction. The synchronous languages Statecharts [22] and Statemate [13] also distinguish internal from external events. Although the descriptions suggest a stack-based semantics, we are not aware of formalizations or more precision for a deeper comparison with Céu.

Like Céu, many other synchronous languages [2, 9, 12, 14, 23] rely on lexical scheduling to preserve determinism. In contrast, in Esterel, the execution order for operations within a reaction is non-deterministic: *"if there is no control dependency, as in* `(call f1()||call f2())`*, the order is unspecified and it would be an error to rely on it"* [6]. For this reason, Esterel, does not support shared-memory concurrency: *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"* [6]. Considering the constant and low-level interactions with the underlying architecture in embedded systems (e.g., direct port manipulation), we believe that it is advantageous to embrace lexical scheduling as part of the language specification as a pragmatic design decision to enforce determinism. However, since Céu statically detects trails not sharing memory, an optimized scheduler could exploit real parallelism in such cases.

Regarding the integration with *C* language-based environments, Céu supports a finalization mechanism for external resources. In addition, Céu also tracks pointers representing resources that cross *C* boundaries and forces the programmer to provide associated finalizers. As far as we know, this extra safety level is unique to Céu among synchronous languages.

6 Conclusion

The programming language Céu aims to offer a concurrent, safe, and realistic alternative to C for embedded soft real-time systems, such as sensor networks and multimedia systems. Céu inherits the synchronous and imperative mindset of Esterel but adopts a simpler semantics with fine-grained execution control, which makes the language fully deterministic. In addition, its stack-based execution for internal events provides a limited but memory-bounded form of subroutines. Céu also provides a finalization mechanism for resources when interacting with the external environment.

In this paper, we proposed a small-step operational semantics for Céu and proved that under it reactions are deterministic, terminate in finite time, and use bounded memory, i.e., that for a given arbitrary timeline of input events, multiple executions of the same program always react in bounded time and arrive at the same final finite memory state.

References

[1] A. Adya et al. Cooperative task management without manual stack management. In *Proceedings of ATEC'02*, pages 289–302. USENIX Association, 2002.

[2] S. Andalam, P. Roop, and A. Girault. Predictable multithreading of embedded applications using PRET-C. In *Proceeding of MEMOCODE'10*, pages 159–168. IEEE, 2010.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, volume 91, pages 64–83, Jan 2003.

[4] G. Berry. Preemption in concurrent systems. In *FSTTCS*, volume 761 of *LNCS*, pages 72–93. Springer, 1993.

[5] G. Berry. *The Constructive Semantics of Pure Esterel (draft version 3)*. Ecole des Mines de Paris and INRIA, 1999.

[6] G. Berry. *The Esterel-V5 Language Primer*. CMA and Inria, Sophia-Antipolis, France, June 2000. Version 5.10, Release 2.0.

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] T. Bourke and A. Sowmya. Delays in esterel. page 55, 2009.

[9] F. Boussinot. Reactive c: An extension of c to program reactive systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[10] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.

[11] R. de Simone, J.-P. Talpin, and D. Potop-Butucaru. The synchronous hypothesis and synchronous languages. In R. Zurawski, editor, *Embedded Systems Handbook*. 2005.

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*, pages 29–42. ACM, 2006.

[13] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[14] M. Karpinski and V. Cahill. High-level application development is realistic for wireless sensor networks. In *Proceedings of SECON'07*, pages 610–619, 2007.

[15] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.

[16] ORACLE. Java thread primitive deprecation. http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html (accessed in Aug-2014), 2011.

[17] G. D. Plotkin. A structural approach to operational semantics. Technical Report 19, Computer Science Departement, Aarhus University, Aarhus, Denmark, 1981.

[18] G. Salvaneschi et al. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of Modularity'13*, pages 25–36. ACM, 2014.

[19] F. Sant'anna, R. Ierusalimschy, N. Rodriguez, S. Rossetto, and A. Branco. The design and implementation of the synchronous language cÉu. *ACM Trans. Embed. Comput. Syst.*, 16(4):98:1–98:26, July 2017.

[20] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM, 2013.

[21] R. Santos, G. Lima, F. Sant'Anna, and N. Rodriguez. Céu-Media: Local Inter-Media Synchronization Using Céu. In *Proceedings of WebMedia'16*, pages 143–150, New York, NY, USA, 2016. ACM.

[22] M. von der Beeck. A comparison of statecharts variants. In *Proceedings of FTRTFT'94*, pages 128–148. Springer, 1994.

[23] R. von Hanxleden. Synccharts in c: a proposal for light-weight, deterministic concurrency. In *Proceedings EMSOFT'09*, pages 225–234. ACM, 2009.