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

1 Introduction

Céu [17, 19] is a Esterel-based [8] 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, 13, 16]. 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%) [19]. Céu has also been used in the context of multimedia systems [20] and games [18].

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 [17]:

- 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.

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. The current 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.

2 Céu

Céu [17, 19] 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 [9]. 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 one *await* statement to an external input event [6, 19]. 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.

Listing 1 shows a compact reference of Céu.

```plaintext
// 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
finalise ⟨⟨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⟩(⟨expr⟩); // calls a C function (id starts with ‘_’)
```

Listing 1. The concrete syntax 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 *finalise* 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.\footnote{The actual implementation of Céu supports a command-line option that accepts a white list of library calls. If a program tries to use a function not in the list, the compiler raises an error.}

```c
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
```

**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 [19]:

(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 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.

![Figure 1. The discrete notion of time in Céu.](attachment:image.png)
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).

```plaintext
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. 3), and assigning to them in parallel trails (ln. 6, 9).

In [a], the two assignments to x can only execute in reactions to different events A and B (ln. 5, 8), which cannot occur simultaneously by definition. Hence, for the sequence of events A→B, x becomes 4, while for B→A, x becomes 3.

In [b], the two assignments to y are simultaneous because they execute in reaction to the same event A. 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 [19]. Nonetheless, the static checks are optional and are not a prerequisite for the deterministic semantics of the language.

### 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, 14].

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.

```plaintext
input void A; 1 input void A;
input void B; 2 // (empty line)
var int x = 1; 3 var int y = 1;
par/or do 4 par/or do
await A; 5 await A;
  x = x + 1; 6 y = y + 1;
with 7 with
  await B; 8 await A;
  x = x * 2; 9 y = y * 2;
end 10 end
```

[a] Accesses to x are never concurrent. [b] Accesses to y are concurrent but still deterministic.

Figure 2. Shared-memory concurrency in Céu: [a] is safe because the trails access x atomically in different reactions; [b] is unsafe because both trails access y in the same reaction.
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 (ln. [a]:5), the pointer represents a local resource (ln. [a]:2) that requires finalization (ln. [a]:7). If Céu receives a pointer from a system call return (ln. [b]:4), the pointer represents an external resource (ln. [b]:2) that requires finalization (ln. [b]:6).

Céu tracks the interaction of system calls with pointers and requires finalization clauses to accompany them. In Figure 3.a, 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 acknowledgment 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 Figure 3.b, 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

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.

## 3 Formal Semantics

In this section, we introduce a reduced, abstract syntax for Céu and present an operational semantics that formalizes the behavior of its programs. The semantics deals only with the control aspects of Céu. Side-effects and system calls are encapsulated in a memory access primitive and are assumed to behave like in conventional imperative languages.

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

<table>
<thead>
<tr>
<th>par/or do</th>
<th>par/or do</th>
</tr>
</thead>
<tbody>
<tr>
<td>var _msg_t msg;</td>
<td>var _FILE* f;</td>
</tr>
<tr>
<td>⟨⋯⟩ // prepare msg</td>
<td>finalize</td>
</tr>
<tr>
<td>finalize</td>
<td>f = _fopen(...);</td>
</tr>
<tr>
<td>_send(&amp;msg);</td>
<td>with</td>
</tr>
<tr>
<td>with</td>
<td>_fclose(f);</td>
</tr>
<tr>
<td>_cancel(&amp;msg);</td>
<td>end</td>
</tr>
<tr>
<td>end</td>
<td>_fwrite(..., f);</td>
</tr>
<tr>
<td>await SEND_ACK;</td>
<td>await A;</td>
</tr>
<tr>
<td>with</td>
<td>_fwrite(..., f);</td>
</tr>
<tr>
<td>⟨⋯⟩</td>
<td>with</td>
</tr>
<tr>
<td>end</td>
<td>⟨⋯⟩</td>
</tr>
<tr>
<td>// (empty line)</td>
<td>end</td>
</tr>
</tbody>
</table>

[a] Local resource finalization.  [b] External resource finalization.

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

### 3.1 Abstract Syntax

The grammar below defines the syntax of a subset of Céu that is sufficient to describe all peculiarities of the language.

\[
P := \text{mem}(id) \mid \text{await}_\text{int}(id) \mid \text{await}_\text{ext}(id) \mid \text{emit}_\text{int}(id) \mid \text{break} \mid \text{if mem}(id) \text{ then } P_1 \text{ else } P_2 \mid P_1 : P_2 \mid \text{loop } P_1 \mid \text{every } id P_1 \mid P_1 \text{ and } P_2 \mid P_2 \text{ or } P_2 \mid \text{fin } P \mid P_1 @\text{loop } P_2 \mid P_1 \text{ and } P_2 \mid P_1 \text{ or } P_2 \mid \text{cancel}(n) \mid \text{canrun}(n) \mid \text{nop}
\]

The mem(id) primitive represents a read or write to the memory location identified by id.2 Following the synchronous hypothesis, mem statements and expressions are considered to be atomic and instantaneous.

Most statements in the abstract language are mapped to their counterparts in the concrete language. The exceptions are the finalization block finP and the @-statements which do not exist in the concrete language. These result from the expansion of the transition rules to be presented.

A further difference between the concrete and abstract languages regards the emit-await pair. In the concrete language, the await can be used as an expression which evaluates to the value stored in the corresponding emitted event, e.g., an “emit a(10)” awakes a “v=await a” setting variable v to 10. Although the abstract emit_int and emit_ext do not support such communication of values, it can be easily simulated: one can use a shared variable to hold the value of an emit_int and access it after the corresponding await_int awakes.

Finally, a “finalize A with B end; C” in the concrete language is equivalent to “A; (fin B) or C” in the abstract language. In the concrete language, A and C execute in sequence, and the finalization code B is implicitly suspended waiting for C to terminate. In the abstract language, “fin B” suspends forever when reached (it is an awaiting statement that never awakes). Hence, we need an explicit or to execute C in parallel, whose termination aborts “fin B”, which finally causes B to execute (by the semantic rules below).

2Although the same symbol id is used in their definition, mem, await_int, await_ext and emit_int do not share identifiers: any identifier is either a variable, an external event, or an internal event.
3.2 Operational Semantics

The operational semantics is a mathematical model that describes how an abstract Céu program reacts to a single external input event, i.e., how starting from this input event the program advances its state until all its trails are blocked waiting for another input event. The formalism we use here is that of small-step operational semantics [15] in which the meaning of programs is defined in terms of transitions of an abstract machine. Each transition transforms a triple consisting of a program \( p \), a stack level \( n \), and an emitted event \( e \) into a possibly different triple, i.e.,

\[
(p, n, e) \rightarrow (p', n', e'),
\]

where \( p, p' \in P \) are abstract-language programs, \( n, n' \in N \) are non-negative integers representing the current stack level, and \( e, e' \in E \cup \{\varepsilon\} \) are the events emitted before and after the transition (both possibly being the empty event \( \varepsilon \)).

We refer to the triples on the left-hand and right-hand sides of symbol \( \rightarrow \) as descriptions (denoted \( \delta \)). The description on the left of \( \rightarrow \) is called the input description, the one on its right is called the output description.

At the beginning of a reaction to an input event \( id \), the input description is initialized with stack level 0 \((n = 0)\) and with the externally emitted event \( e = id \). At the end of a reaction, after an arbitrary but finite number of transitions, the last output description will block with a possibly modified program \( p' \) at stack level 0 and no event emitted:

\[
(p, 0, e) \rightarrow (p', 0, \varepsilon).
\]

We now proceed to give the rules for possible the transitions. We distinguish between two types of transitions: outermost transitions \( \stackrel{\text{out}}{\rightarrow} \) and nested transitions \( \stackrel{\text{nst}}{\rightarrow} \).

3.2.1 Outermost Transitions

The rules \( \text{push} \) and \( \text{pop} \) for \( \stackrel{\text{out}}{\rightarrow} \) transitions are non-recursive definitions that apply to the program as a whole. These are the only rules that manipulate the stack level.

\[
\begin{align*}
\text{push} & \quad e \neq \varepsilon \\
(p, n, e) & \stackrel{\text{out}}{\rightarrow} (\text{bcast}(p, e), n + 1, e) \\
\text{pop} & \quad n > 0 \\
(p, n, e) & \stackrel{\text{out}}{\rightarrow} (p, n - 1, \varepsilon)
\end{align*}
\]

Rule \( \text{push} \) can be applied whenever there is a nonempty event in the input description, and instantly broadcasts the event to the program, which means (i) awaking any active \( \text{await}_{\text{ext}} \) or \( \text{await}_{\text{int}} \) statements (see \( \text{bcast} \) in Figure 4), (ii) creating a nested reaction by increasing the stack level, and, at the same time, (iii) consuming the event (\( e \) becomes \( \varepsilon \)). Rule \( \text{push} \) is the only rule that matches an emitted event and also immediately consumes it.

\[
\text{pop} \quad \text{decreases the stack level by one and can only be applied if the program is blocked (see } \text{isblocked} \text{ in Figure 4) or terminated (} p = \text{@nop} \text{ or } p = \text{break} \text{). This condition ensures that an } \text{emit}_{\text{int}} \text{ only resumes after its internal reaction completes and blocks in the current stack level.}
\]

At the beginning of a reaction, an external event is emitted, which triggers rule \( \text{push} \), immediately raising the stack level to 1. At the end of the reaction, the program will block or terminate and successive applications of rule \( \text{pop} \) will lead to a description with this same program at stack level 0.

3.2.2 Nested Transitions

The \( \stackrel{\text{nst}}{\rightarrow} \) rules are recursive definitions of the form

\[
(p, n, e) \stackrel{\text{nst}}{\rightarrow} (p', n, e).
\]

Nested transitions do not affect the stack level and never have an emitted event as a precondition. The distinction between \( \stackrel{\text{out}}{\rightarrow} \) and \( \stackrel{\text{nst}}{\rightarrow} \) prevents rules \( \text{push} \) and \( \text{pop} \) from matching and, consequently, from inadvertently modifying the current stack level before the nested reaction is complete.

A complete reaction consists of a series of transitions

\[
\langle p, 0, e_{\text{ext}} \rangle \stackrel{\text{push}}{\stackrel{\text{out}}{\rightarrow}} \langle p_{1}, 1, e \rangle \rightarrow \langle p'_{1}, 1, e \rangle \rightarrow \langle p', 0, \varepsilon \rangle.
\]

First, a \( \text{push} \) \( \text{out} \) starts a nested reaction at level 1. Then, a series of alternations between zero or more \( \stackrel{\text{nst}}{\rightarrow} \) transitions (nested reactions) and a single \( \stackrel{\text{out}}{\rightarrow} \) transition (stack operation) takes place. Finally, a last \( \text{pop} \) \( \text{out} \) transition decrements the stack level to 0 and terminates the reaction.

The \( \stackrel{\text{nst}}{\rightarrow} \) rules for atoms are defined as follows:

\[
\begin{align*}
\langle \text{mem}(id), n, e \rangle & \stackrel{\text{nst}}{\rightarrow} \langle \text{@nop}, n, e \rangle & \quad \text{(mem)} \\
\langle \text{emit}_{\text{int}}(id), n, e \rangle & \stackrel{\text{nst}}{\rightarrow} \langle \text{@canrun}(n), n, id \rangle & \quad \text{(emit-int)} \\
\langle \text{@canrun}(n), n, e \rangle & \stackrel{\text{nst}}{\rightarrow} \langle \text{@nop}, n, e \rangle & \quad \text{(can-run)}
\end{align*}
\]

A \( \text{mem} \) operation becomes a \( \text{nop} \) which indicates the memory access (rule \( \text{mem} \)). An \( \text{emit}_{\text{int}}(id) \) generates an event \( id \) and becomes a \( \text{canrun}(n) \) which can only resume at level \( n \) (rule \( \text{emit-int} \)). Since all \( \stackrel{\text{nst}}{\rightarrow} \) rules can only be applied if \( e = \varepsilon \), an \( \text{emit}_{\text{int}} \) inevitably causes rule \( \text{push} \) to execute at the outer level, creating a new level \( n + 1 \) on the stack. Also, with the new stack level, the resulting \( \text{canrun}(n) \) itself cannot transition yet (rule \( \text{can-run} \)), providing the desired stack-based semantics for internal events.

The rules for conditionals and sequences are the following:

\[
\begin{align*}
\text{eval}(\text{mem}(id)) & \quad \text{(if-true)} \\
\langle \text{if mem}(id) \text{ then p else q, n, e} \rangle & \stackrel{\text{nst}}{\rightarrow} \langle p, n, e \rangle \\

\langle \text{if mem}(id) \text{ then p else q, n, e} \rangle & \stackrel{\text{nst}}{\rightarrow} \langle q, n, e \rangle & \quad \text{(if-true)}
\end{align*}
\]

\[
\begin{align*}
\langle \text{if mem}(id) \text{ then p else q, n, e} \rangle & \stackrel{\text{nst}}{\rightarrow} \langle p', n, e \rangle & \quad \text{(if-true)} \\
\langle p ; q, n, e \rangle & \stackrel{\text{nst}}{\rightarrow} \langle p' ; q, n, e \rangle & \quad \text{(seq-adv)} \\
\langle \text{@nop} ; q, n, e \rangle & \stackrel{\text{nst}}{\rightarrow} \langle q, n, e \rangle & \quad \text{(seq-nop)} \\
\langle \text{break} ; q, n, e \rangle & \stackrel{\text{nst}}{\rightarrow} \langle \text{break}, n, e \rangle & \quad \text{(seq-brk)}
\end{align*}
\]

\footnote{We write \( \rightarrow \) to mean \( k \) transitions in sequence, and we write \( \rightarrow \) to mean a finite (possibly zero) number of transitions in sequence.}
Rules **if-true** and **if-false** are the only rules that use `mem` in a way that affects the control flow. Function `eval` evaluates a `mem` expression to a boolean value.

The rules for loops are similar to those for sequences, but use “;” as separators to bind breaks to their enclosing loops:

\[
\begin{align*}
\langle \text{loop } p, n, e \rangle \xrightarrow{nst} \langle \text{loop } p, n, e \rangle & \quad \text{(loop-expd)} \\
\langle q, n, e \rangle \xrightarrow{nst} \langle q', n, e \rangle & \quad \text{(loop-adv)} \\
\langle \text{nop } \text{loop } p, n, e \rangle \xrightarrow{nst} \langle q' \text{ loop } p, n, e \rangle & \quad \text{(loop-nop)} \\
\langle \text{break } \text{loop } p, n, e \rangle \xrightarrow{nst} \langle \text{nop } n, e \rangle & \quad \text{(loop-brk)}
\end{align*}
\]

When a program encounters a loop, it first expands its body in sequence with itself (rule `loop-expd`). Rules `loop-adv` and `loop-nop` are similar to rules `seq-adv` and `seq-nop`, advancing the loop until a `nop` is reached. However, what follows the loop is the loop itself (rule `loop-nop`). Note that if we used “;” as a separator in loops, rules `loop-brk` and `seq-brk` would conflict. Rule `loop-brk` escapes the enclosing loop, transforming everything into a `nop`.

The rules for and and or compositions ensure that their left branch always transition before their right branch:

\[
\begin{align*}
\langle p \text{ and } q, n, e \rangle \xrightarrow{nst} \langle p \text{ and } q, n, e \rangle & \quad \text{(expd)} \\
\langle p \text{ and } q, n, e \rangle \xrightarrow{nst} \langle p', n, e \rangle & \quad \text{(adv1)} \\
\langle p \text{ or } q, n, e \rangle \xrightarrow{nst} \langle p \text{ or } q, n, e \rangle & \quad \text{(expd)} \\
\langle p \text{ or } q, n, e \rangle \xrightarrow{nst} \langle p', n, e \rangle & \quad \text{(adv1)} \\
\langle p \text{ or } q, n, e \rangle \xrightarrow{nst} \langle p' \text{ or } q, n, e \rangle & \quad \text{(adv2)}
\end{align*}
\]

Rules `expd` and `advd` insert a `canrun(n)` at the beginning of the right branch. This ensures that any `emit_int` on the left branch, which eventually becomes a `canrun(n)`, resumes before the right branch starts. The deterministic behavior of the semantics relies on the `isblocked` predicate (see Figure 4) which is used in rules `adv2` and `adv2`. These rules require the left branch `p` to be blocked for the right branch to transition from `q` to `q'`.

In a parallel `and`, if one branch terminates, the composition becomes the other branch (rules `and-nop1` and `and-nop2` below). In a parallel `or`, however, if one branch terminates, the whole composition terminates and `clear` is called to finalize the aborted branch (rules `or-nop1` and `or-nop2`).
Theorem 3.3 (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

$$\delta \xrightarrow{1} \delta_1', \delta \xrightarrow{1} \delta_2'$$

for some $i > 1$, $\delta_1'$ and $\delta_2'$. Then, by Lemma 3.1 or 3.2, depending on whether the first transition is $\xrightarrow{\text{out}}$ or $\xrightarrow{\text{in}}$ (it cannot be both), $\delta_1' = \delta_2'$, and by the induction hypothesis, $\delta_1 = \delta_2$. \hfill $\square$

3.3.2 Termination

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 $\xrightarrow{\text{out}}$ and $\xrightarrow{\text{in}}$.

Definition 3.4. A description $\delta = \langle p, n, e \rangle$ is nested-irreducible iff $e \neq e$ or $p = \@nop$ or $p = \@break$ or $\text{isblocked}(p, n)$.

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

The use of qualifier “irreducible” in Definition 3.4 is justified by Proposition 3.5, which states that if a finite number of applications of $\xrightarrow{\text{in}}$ results in an irreducible description, then that occurs exactly once, at some specific number $i$. The proof of Proposition 3.5 follows directly from the definition of $\xrightarrow{\text{in}}$ by contradiction on the hypothesis that there is such $k \neq i$.

Proposition 3.5. If $\delta \xrightarrow{\text{in}}^k \delta_{\text{nest}}$ then, for all $k \neq i$, there is no $\delta_{\text{nest}}'$ such that $\delta \xrightarrow{\text{in}}^k \delta_{\text{nest}}'$.

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

Lemma 3.6. If $\langle p_1, n, e \rangle \xrightarrow{\text{in}} \langle p_1', n, e' \rangle$, for any $p_2$:

(a) $\langle p_1; p_2, n, e \rangle \xrightarrow{\text{in}} \langle p_1', p_2, n, e' \rangle$.
(b) $\langle p_1 \oplus \text{loop} p_2, n, e \rangle \xrightarrow{\text{in}} \langle p_1' \oplus \text{loop} p_2, n, e' \rangle$.
(c) $\langle p_1 \& p_2, n, e \rangle \xrightarrow{\text{in}} \langle p_1' \& p_2, n, e' \rangle$.
(d) $\langle p_1 \@ \& p_2, n, e \rangle \xrightarrow{\text{in}} \langle p_1' \@ \& p_2, n, e' \rangle$.

If $\langle p_2, n, e \rangle \xrightarrow{\text{in}} \langle p_2', n, e' \rangle$, for any $p_1$ such that $\text{isblocked}(p_1, n)$:

(a) $\langle p_1 \& p_2, n, e \rangle \xrightarrow{\text{in}} \langle p_1' \& p_2, n, e' \rangle$.
(b) $\langle p_1 \@ \& p_2, n, e \rangle \xrightarrow{\text{in}} \langle p_1' \@ \& p_2, n, e' \rangle$.

5We sometimes abbreviate “$p = \@nop$ or $p = \@break$” as “$p = \@nop$, break”.

Figure 4. (i) Function $\text{bcast}$ awakes awaiting trails matching the event by converting $\text{await\_ext}$ and $\text{await\_int}$ to $\@nop$, and by unwinding every statements. (ii) Predicate $\text{isblocked}$ is true only if all branches in parallel are blocked waiting for events, finalization clauses, or certain stack levels. (iii) Function $\text{clear}$ extracts $\text{fin}$ statements in parallel and put their bodies in sequence. In (i), (ii), and (iii), “$\_\_\_$” denotes the omitted cases and “$\xi$” denotes the empty string.

Lemma 3.1. If $\delta \xrightarrow{\text{out}} \delta_1$ and $\delta \xrightarrow{\text{out}} \delta_2$ then $\delta_1 = \delta_2$.

Lemma 3.2. If $\delta \xrightarrow{\text{in}} \delta_1$ and $\delta \xrightarrow{\text{in}} \delta_2$ then $\delta_1 = \delta_2$.

The main result of this section, Theorem 3.3, 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.
The syntactic restriction discussed in Section 2.1 regarding
the body of loops and the restriction mentioned in Section 3.2
about the body of fin statements are formalized in Assumption
3.7 below. These restrictions are essential to prove the
next theorem.

**Assumption 3.7** (Syntactic restrictions).
(a) If $p = \text{fin} p_1$ then $p_1$ contains no occurrences of state-
ments $\text{await}_{\text{ext}}, \text{await}_{\text{int}}, \text{every}, \text{or fin}$. And so, for
any $n$, $\text{clear}(p_1, n, e) \rightarrow_{\text{ext}} \text{nop}(n, e)$.
(b) If $p = \text{loop} p_1$ then all execution paths of $p_1$ contain a
matching break or an await$_{\text{ext}}$. Consequently, for all $n$, there
are $p'$ and $e$ such that $(\text{loop} p_1, n, e) \rightarrow_{\text{ext}} (p', n, e)$,
where $p' = \text{break} @ \text{loop} p_1$ or $\text{isblocked}(p', n)$.

Theorem 3.8 establishes that a finite (possibly zero) num-
ber of $\rightarrow_{\text{ext}}$ transitions eventually leads to a nested-irreducible
description. Hence, for any input description $\delta$, it is always
possible to transform $\delta$ in a nested-irreducible description $\delta'$
by applying to it a sufficiently long sequence of $\rightarrow_{\text{ext}}$ tran-
siptions. The proof of the theorem follows by induction on the
structure of programs (members of set $P$) and depends on
Lemma 3.6 and Assumption 3.7.

**Theorem 3.8.** For any $\delta$ there is a $\delta'_n$ such that $\delta \rightarrow_{\text{ext}} \delta'_n$.

The main result of this section, Theorem 3.15, is similar to
Theorem 3.8 but applies to transitions $\rightarrow$ in general. Before
stating and proving it, we need to characterize irreducible
descriptions in general. This characterization, given in Defi-
nition 3.11, depends on the notions of potency and rank.

**Definition 3.9.** The **potency** of a program $p$ in reaction to
event $e$, denoted $\text{pot}(p,e)$, is the maximum number of emit$_{\text{int}}$
statements that can be executed in a reaction of $p$ to $e$, i.e.,
$$\text{pot}(p,e) = \text{pot}'(\text{broadcast}(p,e)),$$
where $\text{pot}'$ is an auxiliary function that counts the maximum
number of reachable emit$_{\text{int}}$ statements in the program re-
sulting from the broadcast of event $e$ to $p$.

Function $\text{pot}'$ is defined by the following clauses:
(a) $\text{pot}'(\text{emit}_{\text{int}}(e)) = 1$.
(b) $\text{pot}'(\text{if mem(id) then } p_1 \text{ else } p_2) = \max\{\text{pot}'(p_1), \text{pot}'(p_2)\}$.
(c) $\text{pot}'(\text{loop } p_1) = \text{pot}'(p_1)$.
(d) $\text{pot}'(p_1 \text{ and } p_2) = \text{pot}'(p_1) + \text{pot}'(p_2)$.
(e) If $p_1 \neq \text{break, } \text{await}_{\text{ext}}(e)$,
$$\text{pot}'(p_1; p_2) = \text{pot}'(p_1) + \text{pot}'(p_2)$$
$$\text{pot}'(p_1; @ \text{loop } p_2) = \begin{cases} \text{pot}'(p_1) & \text{if (1)} \\
\text{pot}'(p_1) + \text{pot}'(p_2) & \text{otherwise,}
\end{cases}$$
where (1) stands for: "a break or await$_{\text{ext}}$ occurs in all
execution paths of $p_1$".
(f) If $p_1, p_2 \neq \text{break}, \text{pot}'(p_1; \text{ and } p_2) = \text{pot}'(p_1) + \text{pot}'(p_2)$.
(g) If $p_1, p_2 \neq \text{break}$ and $p_1, p_2 \neq \text{nop}$,
$$\text{pot}'(p_1@p_2) = \text{pot}'(p_1) + \text{pot}'(p_2).$$
(h) Otherwise, if none of (a)–(g) applies, $\text{pot}'(\_\_\_\_\_\_) = 0$.

**Definition 3.10.** The **rank** of a description $\delta = \langle p, n, e \rangle$, denoted $\text{rank}(\delta)$, is a pair of nonnegative integers $(i, j)$ such that
$$i = \text{pot}(p,e) \quad \text{and} \quad j = \begin{cases} n \quad \text{if } e = \epsilon \\
n + 1 & \text{otherwise.}
\end{cases}$$

Intuitively, the rank of a description $\delta$ is a measure of the
maximum amount of "work" (transitions) required to trans-
form $\delta$ into an irreducible description, in the following sense.

**Definition 3.11.** A description $\delta$ is irreducible (in symbols, $\delta_i$) iff it is nested-irreducible and its $\text{rank}(\delta) = \langle i, 0 \rangle$, for some $i \geq 0$.

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

The next two lemmas establish that a single application of
$\rightarrow_{\text{out}}$ or $\rightarrow_{\text{ext}}$ either preserves or decreases the rank
of the input description. All rank comparisons assume lexi-
cographic order, i.e., if $\text{rank}(\delta) = \langle i, j \rangle$ and $\text{rank}(\delta') = \langle i', j' \rangle$
then $\text{rank}(\delta) > \text{rank}(\delta')$ iff $i > i'$ or $i = i$ and $j > j'$.
The proof of Lemma 3.12 follows directly from $\text{push}$ and $\text{pop}$
and from Definitions 3.9 and 3.10. The proof of Lemma 3.13,
however, is by induction on the structure of $\rightarrow_{\text{ext}}$ derivations.

**Lemma 3.12.**
(a) If $\delta \rightarrow_{\text{out}} \delta'$ then $\text{rank}(\delta) = \text{rank}(\delta')$.
(b) If $\delta \rightarrow_{\text{ext}} \delta'$ then $\text{rank}(\delta) > \text{rank}(\delta')$.

**Lemma 3.13.** If $\delta \rightarrow_{\text{out}} \delta'$ then $\text{rank}(\delta) \geq \text{rank}(\delta')$.

The next theorem is a generalization of Lemma 3.13 for $\rightarrow_{\text{ext}}$.
Its proof follows from the lemma by induction on $i$.

**Theorem 3.14.** If $\delta \rightarrow_{\text{ext}} \delta'$ then $\text{rank}(\delta) \geq \text{rank}(\delta')$.

We now state and prove the main result of this section,
Theorem 3.15, the termination theorem for $\rightarrow$. The idea of
the proof is that a sufficiently large sequence of $\rightarrow_{\text{ext}}$ and $\rightarrow_{\text{out}}$
transitions eventually decreases the rank of the current de-
scription until an irreducible description is reached. This
irreducible description is the final result of the reaction.

**Theorem 3.15** (Termination). For any $\delta$, there is a $\delta'_n$ such
that $\delta \rightarrow^* \delta'_n$.

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

For the basis, suppose $(i, j) = (0, 0)$. Then $\delta$ cannot be advanced
by $\rightarrow_{\text{ext}}$, as $j = 0$ implies $e = \epsilon$ and $n = 0$. If $\delta$ is nested-
irreducible, the theorem is trivially true, as $\delta = 0 \rightarrow_{\text{ext}} \delta_n$ and $\delta_0$.
If $\delta$ is not nested-irreducible then, by Theorem 3.8, $\delta \rightarrow_{\text{ext}} \delta'_n$,
for some $\delta'_n$. By Theorem 3.14, $\text{rank}(\delta) \geq \text{rank}(\delta')$, which
implies $\text{rank}(\delta') = \langle 0, 0 \rangle$, and so $\delta'_n$. 

9
For the inductive step, suppose \((i, j) \succ (0, 0)\). Then, depending on whether or not \(\delta\) is nested-irreducible, there are two cases.

Case 1. \(\delta\) is nested-irreducible. If \(j = 0\), by Definition 3.11, \(\delta_n\), and so \(\delta \rightarrow \delta_n\). If \(j > 0\), there are two subcases:

Case 1.1. \(e \neq \varepsilon\). Then, by push and by Theorem 3.8, there are \(\delta'_1\) and \(\delta'_n = (p', n + 1, e')\) such that \(\delta \xrightarrow{\text{push}} \delta'_1 \xrightarrow{\text{stub}} \delta'_n\). Thus, by Lemma 3.12 and by Theorem 3.14,

\[
\text{rank}(\delta) = \text{rank}(\delta'_1) = (i, j) \geq \text{rank}(\delta'_1) = (i', j') .
\]

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 \(\text{emit}_n\)(\(e'\)) was consumed by the nested transitions. Thus, \(\text{rank}(\delta) > \text{rank}(\delta')\). By the induction hypothesis, \(\delta' \rightarrow \delta'_o\), for some \(\delta'_o\). Therefore, \(\delta \rightarrow \delta'_o\).

Case 1.2. \(e = \varepsilon\). Then, as \(j > 0\), \(\delta \xrightarrow{\text{pop}_o} \delta'\), for some \(\delta'\). By Lemma 3.12, \(\text{rank}(\delta) > \text{rank}(\delta')\). Hence, by the induction hypothesis, \(\delta' \rightarrow \delta'_o\), for some \(\delta'_o\). And so, \(\delta \rightarrow \delta'_o\).

Case 2. \(\delta\) is not nested-irreducible. Then \(e = \varepsilon\) and, by Theorems 3.8 and 3.14, there is a \(\delta'_n\) such that \(\delta \xrightarrow{\text{stub}} \delta'_n\) with \(\text{rank}(\delta) \geq \text{rank}(\delta'_n)\). The rest of this proof is similar to that of Case 1 above. \(\Box\)

3.3.3 Memory Boundedness

As Cēu has no mechanism for heap allocation, unbounded iteration, or general recursion, the maximum memory usage of a given Cēu 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 execution of program \(p\) to external event \(e\) corresponds to \(\text{pot}(p, e)\), 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 \(\text{max}\{\text{pot}(p, e_1), \ldots, \text{pot}(p, e_n)\}\) events.

4 Related Work

Cēu follows the lineage of imperative synchronous languages initiated by Esterel [8]. These languages typically define time as a discrete sequence of logical “ticks” in which multiple simultaneous input events can be active [17]. The presence of multiple inputs requires careful static analysis to detect and reject programs with causality cycles and schizophrenia problems [5]. In contrast, Cēu 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 Cēu, 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 [21] and Statemate [11] 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 Cēu.

Like Cēu, many other synchronous languages [2, 7, 10, 12, 22] 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 dependence, as in (call \(t_1())\)|call \(t_2())\), 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 cannot 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 Cēu statically detects traits not sharing memory, an optimized scheduler could exploit real parallelism in such cases.

Regarding the integration with C language-based environments, Cēu supports a finalization mechanism for external resources. In addition, Cēu 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 Cēu among synchronous languages.

5 Conclusion

The programming language Cēu aims to offer a concurrent, safe, and realistic alternative to C for embedded soft real-time systems, such as sensor networks and multimedia systems. Cēu 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. Cēu also provides a finalization mechanism for resources when interacting with the external environment.

In this paper, we proposed a small-step operational semantics for Cēu 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.

A Detailed Proofs

Lemma 3.1. If \(\delta_1 \xrightarrow{\text{out}} \delta_2\) and \(\delta_2 \xrightarrow{\text{out}} \delta_2\) then \(\delta_1 = \delta_2\).
**Proof.** The lemma is vacuously true if δ cannot be advanced by transitions. Suppose that is not the case and let δ = (p, n, e), δ1 = (p1, n1, e1) and δ2 = (p2, n2, e2). Then, there are two possibilities.

**Case 1.** e = e. Both transitions are applications of push. Hence p1 = p2 = bcast(p, e), n1 = n2 = n + 1, and e1 = e2 = e.

**Case 2.** e = e. Both transitions are applications of pop. Hence p1 = p2 = p, n1 = n2 = n − 1, and e1 = e2 = e. □

**Lemma 3.2.** If δ → nst δ1 and δ → nst δ2 then δ1 = δ2.

**Proof.** By induction on the structure of → nst derivations. The lemma is vacuously true if δ cannot be advanced by → nst transitions. Suppose that is not the case and let δ = (p, n, e), δ1 = (p1, n1, e1) and δ2 = (p2, n2, e2). Then, by the hypothesis of the lemma, there are derivations π1 and π2 such that

π1 + (p, n, e) → nst (p1, n1, e1)

π2 + (p, n, e) → nst (p2, n2, e2)

i.e., the conclusion of π1 is (p, n, e) → nst (p1, n1, e1) and the conclusion of π2 is (p, n, e) → nst (p2, n2, e2).

By definition of → nst, we have that e = e and n1 = n2 = n. It remains to be shown that p1 = p2 and e1 = e2.

Depending on the structure of program p, the following cases are possible. (Note that p cannot be an await ext, await_int, break, every, fin, or @nop statement as there is no rule to transition such programs.)

**Case 1.** p = mem(id). Then derivations π1 and π2 are instances of rule mem, i.e., their conclusions are obtained by an application of this rule. Hence p1 = p2 = @nop and e1 = e2 = e.

**Case 2.** p = emit_int(e'). Then π1 and π2 are instances of emit_int. Hence p1 = p2 = @canrun(n) and e1 = e2 = e'.

**Case 3.** p = @canrun(n). Then π1 and π2 are instances of canrun. Hence p1 = p2 = @nop and e1 = e2 = e.

**Case 4.** p = if mem(id) then p' else p". There are two subcases.

**Case 4.1.** eval(mem(id)). Then π1 and π2 are instances of if-true. Hence p1 = p2 = p' and e1 = e2 = e.

**Case 4.2.** ¬ eval(mem(id)). Then π1 and π2 are instances of if-false. Hence p1 = p2 = p" and e1 = e2 = e.

**Case 5.** p = p'; p". There are three subcases.

**Case 5.1.** p' = @nop. Then π1 and π2 are instances of seq-nop. Hence p1 = p2 = p' and e1 = e2 = e.

**Case 5.2.** p' = break. Then π1 and π2 are instances of seq-brk. Hence p1 = p2 = break and e1 = e2 = e.

**Case 5.3.** p' ≠ @nop, break. Then π1 and π2 are instances of seq-adv. Thus there are derivations π1 and π2 such that

π1 + (p', n, e) → nst (p', n, e')

π2 + (p', n, e) → nst (p', n, e'')

for some p', p', e', and e". By the induction hypothesis, p' = p' and e' = e'. Hence p1 = p1': p" = p" = p2 and e1 = e1 = e' = e".

**Case 6.** p = loop p'. Then π1 and π2 are instances of loop-expd. Hence p1 = p2 = p'@loop p' and e1 = e2 = e.

**Case 7.** p = p'@loop p". There are three subcases.

**Case 7.1.** p' = @nop. Then π1 and π2 are instances of loop-nop. Hence p1 = p2 = 1oop p' and e1 = e2 = e.

**Case 7.2.** p' = break. Then π1 and π2 are instances of loop-brk. Hence p1 = p2 = @nop and e1 = e2 = e.

**Case 7.3.** p' ≠ @nop, break. Then π1 and π2 are instances of loop-adv. Thus there are derivations π1 and π2 such that

π1 + (p', n, e) → nst (p', n, e')

π2 + (p', n, e) → nst (p', n, e'')

for some p', p', e', and e". By the induction hypothesis, p' = p' and e' = e'. Hence p1 = p1': p" = p" = p2 and e1 = e1 = e' = e2.

**Case 8.** p = p' and p". Then π1 and π2 are instances of and-expd. Hence p1 = p2 = p'and(@canrun(n); p") and e1 = e2 = e.

**Case 9.** p = p' and p". There are two subcases.

**Case 9.1.** ¬ isblocked(p', n). There are three subcases.

**Case 9.1.1.** p' = @nop. Then π1 and π2 are instances of and-nop1. Hence p1 = p2 = p" and e1 = e2 = e.

**Case 9.1.2.** p' = break. Then π1 and π2 are instances of and-brk1. Hence p1 = p2 = clear(p''); break and e1 = e2 = e.

**Case 9.1.3.** p' ≠ @nop, break. Then π1 and π2 are instances of and-adv1. Thus there are derivations π1 and π2 such that

π1 + (p', n, e) → nst (p', n, e')

π2 + (p', n, e) → nst (p', n, e'')

for some p', p', e', and e". By the induction hypothesis, p' = p' and e' = e'. Hence p1 = p1: p" = p" = p2 and e1 = e1 = e' = e2.

**Case 9.2.** isblocked(p', n). There are three subcases.

**Case 9.2.1.** p" = @nop. Then π1 and π2 are instances of and-nop2. Hence p1 = p2 = p' and e1 = e2 = e.

**Case 9.2.2.** p" = break. Then π1 and π2 are instances of and-brk2. Hence p1 = p2 = clear(p''); break and e1 = e2 = e.

**Case 9.2.3.** p" ≠ @nop, break. Then π1 and π2 are instances of and-adv2. Thus there are derivations π1 and π2 such that

π1 + (p', n, e) → nst (p', n, e')

π2 + (p', n, e) → nst (p', n, e'')
for some \( p_1', p_2', e_1', e_2' \). By the induction hypothesis, \( p_1'' = p_2'' \) and \( e_1'' = e_2'' \). Hence \( p_1 = p_2' \) and \( p_1'' = p'' \) and \( p_2'' = p_2' \) and \( e_1 = e_2' = e_2 = e \).

Case 10. \( p = p' \) or \( p'' \). Then \( p_1 \) and \( p_2 \) are instances of \( \text{or-expd} \). Hence \( p_1 = p_2 = \text{or}(\text{canrun}(n); p') \) and \( e_1 = e_2 = e \).

Case 11. \( p = p' \) or \( p'' \). There are two subcases.

Case 11.1. \( \neg \text{isblocked}(p, n) \). There are three subcases.

Case 11.1.1. \( p' = \text{nop} \). Then \( p_1 \) and \( p_2 \) are instances of \( \text{or-nop1} \). Hence \( p_1 = p_2 = \text{clear}(p') \) and \( e_1 = e_2 = e \).

Case 11.1.2. \( p' \neq \text{nop} \). Break. Similar to Case 9.1.2.

Case 11.1.3. \( p' \neq \text{nop} \). Break. Similar to Case 9.1.3.

Case 11.2. \( \text{isblocked}(p', n) \). There are three subcases.

Case 11.2.1. \( p'' = \text{nop} \). Then \( p_1 \) and \( p_2 \) are instances of \( \text{or-nop1} \). Hence \( p_1 = p_2 = \text{clear}(p') \) and \( e_1 = e_2 = e \).

Case 11.2.2. \( p'' = \text{break} \). Similar to Case 9.2.2.

Case 11.2.3. \( p'' \neq \text{nop} \). Break. Similar to Case 9.2.3.

\( \square \)

**Theorem 3.3** (Determinism). \( \delta \xrightarrow{i} \delta_1 \) and \( \delta \xrightarrow{j} \delta_2 \) implies \( \delta_1 = \delta_2 \).

**Proof.** By induction on \( i \). The theorem is trivially true if \( i = 0 \) and follows directly from Lemmas 3.1 and 3.2 for \( i = 1 \). Suppose

\[
\delta \xrightarrow{1} \delta_1', \quad \delta_1' \xrightarrow{1} \delta_1, \quad \text{and} \quad \delta \xrightarrow{1} \delta_2', \quad \delta_2' \xrightarrow{1} \delta_2,
\]

for some \( i > 1 \) and \( \delta_1', \delta_2' \). There are two possibilities.

Case 1. \( \delta \xrightarrow{1} \delta_1', \delta_1' \xrightarrow{1} \delta_1, \) and \( \delta \xrightarrow{1} \delta_2', \delta_2' \xrightarrow{1} \delta_2, \) and by the induction hypothesis, \( \delta_1 = \delta_2 \).

Case 2. \( \delta \xrightarrow{1} \delta_1', \delta_1' \xrightarrow{1} \delta_1, \) and \( \delta \xrightarrow{1} \delta_2', \delta_2' \xrightarrow{1} \delta_2, \) and by the induction hypothesis, \( \delta_1 = \delta_2 \). \( \square \)

**Proposition 3.5.** If \( \delta \xrightarrow{i} \delta''_\text{nest} \) then, for all \( k \neq i \), there is no \( \delta''_\text{nest} \) such that \( \delta \xrightarrow{k} \delta''_\text{nest} \).

**Proof.** By contradiction. Suppose that there is such \( k \). Let \( \delta \xrightarrow{k} \delta''_\text{nest} \), for some \( k \geq 0 \). There are two cases.

Case 1. Suppose there are \( k > i \) and \( \delta''_\text{nest} \) such that \( \delta \xrightarrow{k} \delta''_\text{nest} \). Then, by definition of \( \delta''_\text{nest} \),

\[
\delta \xrightarrow{i} \delta' \xrightarrow{i+1} \delta'' \xrightarrow{i+2} \cdots \xrightarrow{k} \delta''_\text{nest} \xrightarrow{k} \delta''.
\]

Since \( \delta'' = \langle p', n, e' \rangle \) is nested-irreducible, \( e' = e = e' = p = \text{nop}, \text{break}, \text{or-isblocked}(p', n) \). In any of these cases, by the definition of \( \delta''_\text{nest} \), there is no \( \delta''_\text{nest} \) such that \( \delta'' \xrightarrow{0} \delta''_\text{nest} \), which contradicts (1). Therefore, no such \( k \) can exist.

Case 2. Suppose there are \( k < i \) and \( \delta''_\text{nest} \) such that \( \delta \xrightarrow{k} \delta''_\text{nest} \). Then, since \( k > i \), by Case 1, \( \delta''_\text{nest} \) cannot exist, which is absurd. Therefore, the assumption that there is such \( k \) is false. \( \square \)

**Lemma 3.6.** If \( \langle p_1, n, e \rangle \xrightarrow{i} \langle p_1', n, e' \rangle \), for any \( p_2 \):

\[
(a) \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1'; p_2, n, e' \rangle.
\]

\( \square \)

\( \langle p_1 \times (p_2, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \times (p_2, n, e) \rangle. \)

\( \langle p_1 \circ (p_2, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \circ (p_2, n, e) \rangle. \)

\( \langle p_1 \times (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \times (p_2; p_3, n, e) \rangle. \)

\( \langle p_1 \circ (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \circ (p_2; p_3, n, e) \rangle. \)

\( \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\( \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\( \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\( (a) \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle, \)

\( (b) \langle p_1 \times (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \times (p_2; p_3, n, e) \rangle. \)

\( (c) \langle p_1 \circ (p_2, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \circ (p_2, n, e) \rangle. \)

\( (d) \langle p_1 \times (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \times (p_2; p_3, n, e) \rangle. \)

\( (e) \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\( (f) \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\( \langle p_1 \times (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \times (p_2; p_3, n, e) \rangle. \)

\( \langle p_1 \circ (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \circ (p_2; p_3, n, e) \rangle. \)

\( \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\( \langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\(<\langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle, \)

\(<\langle p_1 \times (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \times (p_2; p_3, n, e) \rangle. \)

\(<\langle p_1 \circ (p_2; p_3, n, e) \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', \circ (p_2; p_3, n, e) \rangle. \)

\(<\langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)

\(<\langle p_1; p_2, n, e \rangle \xrightarrow{i \mid \text{nat}} \langle p_1', p_2, n, e' \rangle. \)
Case 3. \( p = \@\text{can-run}(n) \). Then, by \textit{can-run},
\[
\langle \@\text{can-run}(n), n, e \rangle \circ_{\text{nst}}^1 \langle @\text{nop}, n, e \rangle_{\text{nst}}.
\]
Case 4. \( p = \text{if mem}(id) \) then \( p' \) else \( p'' \). There are two subcases.

Case 4.1. \( \text{eval}(\text{mem}(id)) \). Then, by \textbf{if-true} and by the induction hypothesis, there is a \( \delta' \) such that
\[
\langle \text{if mem}(id) \text{ then } p' \text{ else } p'', n, e \rangle \circ_{\text{nst}}^1 \langle p', n, e \rangle \circ_{\text{nst}}^1 \delta'_{\text{nst}}.
\]
Case 4.2. \( \neg \text{eval}(\text{mem}(id)) \). Similar to Case 4.1.

Case 5. \( p = p'; p'' \). There are three subcases.

Case 5.1. \( p' = @\text{nop} \). Then, by \textit{seq-nop} and by the induction hypothesis, there is a \( \delta' \) such that
\[
\langle @\text{nop}; p'', n, e \rangle \circ_{\text{nst}}^1 \langle p'', n, e \rangle \circ_{\text{nst}}^1 \delta'_{\text{nst}}.
\]
Case 5.2. \( p' = \text{break} \). Then, by \textit{seq-brk},
\[
\langle \text{break}; p'', n, e \rangle \circ_{\text{nst}}^1 \langle \text{break}; n, e \rangle_{\text{nst}}.
\]
Case 5.3. \( p' \neq @\text{nop}; \text{break} \). Then, by the induction hypothesis, there are \( p'_1 \) and \( e \) such that
\[
\langle p', n, e \rangle \circ_{\text{nst}}^1 \langle p'_1, n, e \rangle_{\text{nst}}.
\]

By item (a) of Lemma 3.6,
\[
\langle p'; p'', n, e \rangle \circ_{\text{nst}}^1 \langle p'_1; p'', n, e \rangle . \tag{8}
\]

It remains to be shown that \( \langle p'_1; p'', n, e \rangle \) is nested-irreducible.

Case 5.3.1. \( e \neq \ast \). Then, by the definition of \#nst, description \( \langle p'_1; p'', n, e \rangle \) is nested-irreducible.

Case 5.3.2. \( p'_1 = @\text{nop} \). From (8),
\[
\langle p'; p'', n, e \rangle \circ_{\text{nst}}^1 \langle @\text{nop}; p'', n, e \rangle .
\]
From this point on, case is similar to Case 5.1.

Case 5.3.3. \( p'_1 = \text{break} \). From (8),
\[
\langle p'; p'', n, e \rangle \circ_{\text{nst}}^1 \langle \text{break}; p'', n, e \rangle .
\]
From this point on, case is similar to Case 5.2.

Case 5.3.4. \( \text{isblocked}(p'_1, n) \). Then, by definition,
\[
\text{isblocked}(p'_1; p'', n) = \text{isblocked}(p'_1, n) = \text{true}.
\]
Hence from (8) and by the definition \#nst, description \( \langle p'_1; p'', n, e \rangle \) is nested-irreducible.

Case 6. \( p = \text{loop } p' \). Then, by item (b) of Assumption 3.7,
\[
\langle \text{loop } p', n, e \rangle \circ_{\text{nst}}^1 \langle p'_1, n, e \rangle . \tag{9}
\]
for some \( e \) and \( p'_1 \) such that either \( p'_1 = \text{break@loop } p' \) or \text{isblocked}(p'_1, n).

Case 6.1. \( p'_1 = @\text{loop } p' \). From (9), by \textit{loop-brk},
\[
\langle \text{loop } p', n, e \rangle \circ_{\text{nst}}^1 \langle \text{break@loop } p', n, e \rangle \circ_{\text{nst}}^1 \langle @\text{nop}, n, e \rangle_{\text{nst}}.
\]

Case 6.2. \( \text{isblocked}(p'_1, n) \). Hence from (9) and by the definition of \#nst, \( \langle p'_1, n, e \rangle_{\text{nst}} \).

Case 7. \( p = p' @\text{loop } p'' \). There are three subcases.

Case 7.1. \( p' = @\text{loop} \). Then, by \textit{loop-nop},
\[
\langle @\text{loop } p'', n, e \rangle \circ_{\text{nst}}^1 \langle @\text{loop } p'', n, e \rangle_{\text{nst}}.
\]
From this point on, this case is similar to Case 6.

Case 7.2. \( p' = \text{break} \). Then, by \textit{loop-brk},
\[
\langle \text{break@loop } p'', n, e \rangle \circ_{\text{nst}}^1 \langle @\text{loop } p'', n, e \rangle_{\text{nst}}.
\]
Case 7.3. \( p' \neq @\text{nop}; \text{break} \). Then, by the induction hypothesis, there are \( p'_1 \) and \( e \) such that
\[
\langle p', n, e \rangle \circ_{\text{nst}}^1 \langle p'_1, n, e \rangle_{\text{nst}} .
\]
By item (b) of Lemma 3.6,
\[
\langle p' @\text{loop } p'', n, e \rangle \circ_{\text{nst}}^1 \langle p'_1 @\text{loop } p'', n, e \rangle .
\]
It remains to be shown that \( \langle p'_1 @\text{loop } p'', n, e \rangle \) is nested-irreducible. The rest of this proof is similar to that of Case 5.3.

Case 8. \( p = p' \) and \( p'' \). Then, by \textit{and-expd},
\[
\langle p' \) and \( p'', n, e \rangle \circ_{\text{nst}}^1 \langle @\text{and} @\text{can-run}(n); p''', n, e \rangle .
\]
From this point on, this case is similar to Case 9.

Case 9. \( p = p' \) and \( p''' \). There are two subcases.

Case 9.1. \( \neg \text{isblocked}(p'_1, n) \). There are three subcases.

Case 9.1.1. \( p' = @\text{nop} \). Then, by \textit{and-nop} and by the induction hypothesis, there is a \( \delta' \) such that
\[
\langle @\text{and} @\text{nop}; p''' n, e \rangle \circ_{\text{nst}}^1 \langle @\text{and} @\text{nop}; p''' n, e \rangle \circ_{\text{nst}}^1 \delta'_{\text{nst}} .
\]
Case 9.1.2. \( p' = \text{break} \). Then, by \textit{and-brk}.
\[
\langle \text{break@and } p''' n, e \rangle \circ_{\text{nst}}^1 \langle \text{clear}(p'); \text{break}; n, e \rangle . \tag{10}
\]
From (10), by item (a) of Assumption 3.7 and by \textit{seq-nop},
\[
\langle \text{clear}(p'); \text{break}; n, e \rangle \circ_{\text{nst}}^1 \langle @\text{and} @\text{nop}; p''' n, e \rangle \circ_{\text{nst}}^1 \langle \text{break}; n, e \rangle_{\text{nst}} .
\]
Case 9.1.3. \( p' \neq @\text{nop} \). Then, by the induction hypothesis, there are \( p'_1 \) and \( e \) such that
\[
\langle p', n, e \rangle \circ_{\text{nst}}^1 \langle p'_1, n, e \rangle_{\text{nst}} .
\]
By item (c) of Lemma 3.6,
\[
\langle p' @\text{and } p'''; n, e \rangle \circ_{\text{nst}}^1 \langle p'_1 @\text{and } p''', n, e \rangle .
\]
It remains to be shown that \( \langle p'_1 @\text{and } p'''', n, e \rangle \) leads to an nested-irreducible description. There are four possibilities following from the fact that the simpler \( \langle p'_1, n, e \rangle \) is nested-irreducible.

1. If \( e \neq \ast \) then, by definition, \( \langle p'_1 @\text{and } p'''', n, e \rangle_{\text{nst}} \).
2. If \( p'_1 = @\text{nop} \), this case is similar to Case 9.1.1.
3. If \( p'_1 = \text{break} \), this case is similar to Case 9.1.2.
4. If isblocked(p′, n), this case is similar to Case 9.2.
   Case 9.2. isblocked(p′, n). There are three subcases.
   Case 9.2.1. p′′ = @nop. Then, by and-nop2,
   \[ \langle p′ \text{ and } @nop, n, e \rangle \xrightarrow{1} \text{nt} \langle p′′, n, e \rangle_{\text{nest}}. \]
   Case 9.2.2. p′′ = break. Then, by and-brk2,
   \[ \langle p′ \text{ and break, n, e} \rangle \xrightarrow{1} \langle \text{clear}(p′); \text{break, n, e} \rangle. \]
   From this point on, this case is similar to Case 9.1.2.
   Case 9.2.3. p′′ ≠ @nop, break. Then, by the induction hypothesis, there are p′″ and e such that
   \[ \langle p′′, n, e \rangle \xrightarrow{1} \langle p′″, n, e \rangle_{\text{nest}}. \]
   By item (a) of Lemma 3.6,
   \[ \langle p′ \text{ and } p′″, n, e \rangle \xrightarrow{1} \langle p′ \text{ or } p′″, n, e \rangle. \]
   It remains to be show that (p′ and p′″, n, e) leads to an nested-irreducible description. There are four possibilities following from the fact that the simpler (p′″, n, e) is nested-irreducible.
   1. If e ≠ e then, by definition, \( \langle p′ \text{ and } p′″, n, e \rangle = \langle p′ \text{ or } p′″, n, e \rangle \).
   2. If p′″ = @nop, this case is similar to Case 9.2.1.
   3. If p′″ = break, this case is similar to Case 9.2.2.
   4. If isblocked(p′″, n) then, as both sides are blocked, by definition, \( \langle p′ \text{ and } p′″, n, e \rangle_{\text{nest}} \).

Case 10. p = p′ or p′″. Then, by or-expd,
\[ \langle p′ \text{ or } p′″, n, e \rangle \xrightarrow{1} \langle p′ \text{ or } \pi \rangle \langle \text{can-run}(n); p′″, n, e \rangle. \]
From this point on, this case is similar to Case 11.
Case 11. p = p′ or p′″. There are two subcases.
Case 11.1. ¬ isblocked(p′, n). There are three subcases.
   Case 11.1.1. p′ = @nop. Then, by or-nop1,
   \[ \langle @nop \text{ or } p′″, n, e \rangle \xrightarrow{1} \langle \text{clear}(p′″), n, e \rangle. \]
   From (11), by item (a) Assumption 3.7.
   \[ \langle \text{clear}(p′″), n, e \rangle \xrightarrow{1} \langle @nop, n, e \rangle_{\text{nest}}. \]
   Case 11.1.2. p′ = break. Similar to Case 9.1.2.
   Case 11.1.3. p′ ≠ @nop, break. Similar to Case 9.1.3.
Case 11.2. isblocked(p′, n). There are three subcases.
   Case 11.2.1. p′″ = @nop. Then, by or-nop2,
   \[ \langle p′ \text{ or } @nop, n, e \rangle \xrightarrow{1} \langle \text{clear}(p′), n, e \rangle. \]
   From (12), by item (a) of Assumption 3.7 and by definition of clear,
   \[ \langle \text{clear}(p′), n, e \rangle \xrightarrow{1} \langle @nop, n, e \rangle_{\text{nest}}. \]
   Case 11.2.2. p′″ = break. Similar to Case 9.2.2.
   Case 11.2.3. p′″ ≠ @nop, break. Similar to Case 9.2.3.

Lemma 3.12.
Case 5. \( p = p_1; p_2 \). There are three subcases.

Case 5.1. \( p_1 = @\text{nop} \). Then \( \pi \) is an instance of seq-nop.
Hence \( p' = p_2 \) and \( e' = \varepsilon \). Thus
\[
\text{rank}(\delta) = \langle \text{pot}'(p_1) + \text{pot}'(p_2), n \rangle \geq \langle \text{pot}'(p_2), n \rangle = \text{rank}(\delta') .
\]

Case 5.2. \( p_1 = \text{break} \). Then \( \pi \) is an instance of seq-brk.
Hence \( p' = p_1 \) and \( e' = \varepsilon \). Thus
\[
\text{rank}(\delta) = \text{rank}(\delta') = \langle 0, n \rangle .
\]

Case 5.3. \( p_1 \neq @\text{nop} \), \( \text{break} \). Then \( \pi \) is an instance of seq-adv.
Hence there is a derivation \( \pi' \) such that
\[
\pi' \overset{\pi_1, n, \varepsilon}{\rightarrow}^* \langle p'_1, n, e'_1 \rangle ,
\]
for some \( p'_1 \) and \( e'_1 \). Thus \( p' = p'_1; p_2 \) and \( e' = e'_1 \). By the induction hypothesis,
\[
\text{rank}(\langle p_1, n, e \rangle) \geq \text{rank}(\langle p'_1, n, e'_1 \rangle) . \tag{13}
\]

Case 6. \( p = \text{loop } p_1 \). Then \( \pi \) is an instance of loop-expd.
Hence \( p' = p_1 \text{loop } p_1 \) and \( e' = \varepsilon \). By item (b) of Assumption 3.7, all execution paths of \( p_1 \) contain at least one occurrence of break or await\text{ext}. Thus, by condition (\dagger) in Definition 3.9,
\[
\text{rank}(\delta) = \text{rank}(\delta') = \langle \text{pot}'(p_1), n \rangle .
\]

Case 7. \( p = p_1 \text{loop } p_2 \). There are three subcases.

Case 7.1. \( p_1 = @\text{nop} \). Similar to Case 5.1.

Case 7.2. \( p_1 = \text{break} \). Similar to Case 5.2.

Case 7.3. \( p_1 \neq @\text{nop} \), \( \text{break} \). Then \( \pi \) is an instance of loop-adv.
Hence there is a derivation \( \pi' \) such that
\[
\pi' \overset{\pi_1, n, \varepsilon}{\rightarrow}^* \langle p'_1, n, e'_1 \rangle ,
\]
for some \( p'_1 \) and \( e'_1 \). Thus \( p' = p'_1 \text{loop } p_2 \) and \( e' = e'_1 \). There are two subcases.

Case 7.3.1. \( \text{pot}'(p) = \text{pot}'(p_1) \). Then every execution path of \( p_1 \) contains a break or await\text{ext} statement. A single \( \text{wait} \) cannot terminate the loop, since \( p_1 \neq \text{break} \), nor can it consume an await\text{ext}, which means that all execution paths in \( p'_1 \) still contain a break or await\text{ext}. Hence \( \text{pot}'(p') = \text{pot}'(p'_1) \). The rest of this proof is similar to that of Case 5.3.

Case 7.3.2. \( \text{pot}'(p) = \text{pot}'(p_1) + \text{pot}'(p_2) \). Then some execution path in \( p_1 \) does not contain a break or await\text{ext} statement. Since \( p_1 \neq @\text{nop} \), a single \( \text{wait} \) cannot restart the loop, which means that \( p'_1 \) still contain some execution path in which a break or await\text{ext} does not occur. Hence \( \text{pot}'(p') = \text{pot}'(p'_1) + \text{pot}'(p_2) \). The rest of this proof is similar to that of Case 5.3.

Case 8. \( p = p_1 \text{ and } p_2 \). Then \( \pi \) is an instance of and-expd.
Hence \( p' = p_1 \text{ and } (\text{canrun}(n); p_2) \) and \( e' = \varepsilon \). Thus
\[
\text{rank}(\delta) = \text{rank}(\delta') = \langle \text{pot}'(p_1) + \text{pot}'(p_2), n \rangle .
\]

Case 9. \( p = p_1 \text{ and } p_2 \). There are two subcases. stack level n.

Case 9.1. \( \sim \text{isblocked}(p_1, n) \). There are three subcases.

Case 9.1.1. \( p_1 = @\text{nop} \). Then \( \pi \) is an instance of and-nop1. Hence \( p' = p_2 \) and \( e' = \varepsilon \). Thus
\[
\text{rank}(\delta) = \text{rank}(\delta') = \langle 0 + \text{pot}'(p_2), n \rangle .
\]

Case 9.1.2. \( p_1 = \text{break} \). Then \( \pi \) is an instance of and-brk1. Hence \( p' = \text{clear}(p_2) \); break and \( e' = \varepsilon \). By item (a) of Assumption 3.7 and by the definition of clear, \( \text{clear}(p_2) \) does not contain \( \text{emt}\text{int} \), statements. Thus
\[
\text{rank}(\delta) = \text{rank}(\delta') = \langle 0, n \rangle .
\]

Case 9.1.3. \( p_1 \neq @\text{nop} \), \( \text{break} \). Then \( \pi \) is an instance of \( \sim \text{and-adv} \). As \( p_1 \neq \text{break} \) and \( p_2 \neq \text{break} \) (otherwise and-brk2 would have taken precedence), the rest of this proof is similar to that of Case 5.3.

Case 9.2. \( \text{isblocked}(p_1, n) \). Similar to Case 9.1.

Case 10. \( p = p_1 \text{ or } p_2 \). Then \( \pi \) is an instance of or-expd.
Hence \( p' = p_1 \text{ or } (\text{canrun}(n); p_2) \) and \( e' = \varepsilon \). Thus
\[
\text{rank}(\delta) = \text{rank}(\delta') = \langle \text{pot}'(p_1) + \text{pot}'(p_2), n \rangle .
\]

Case 11. \( p = p_1 \text{ or } p_2 \). There are two subcases.

Case 11.1. \( \sim \text{isblocked}(p_1, n) \). There are three subcases.

Case 11.1.1. \( p_1 = @\text{nop} \). Then \( \pi \) is an instance of or-nop1. Hence \( p' = \text{clear}(p_2) \) and \( e' = \varepsilon \). By item (a) of Assumption 3.7 and by the definition of clear, \( p' \) does not contain \( \text{emt}\text{int} \) statements. Thus
\[
\text{rank}(\delta) = \text{rank}(\delta') = \langle 0, n \rangle .
\]

Case 11.1.2. \( p_1 = \text{break} \). Similar to Case 9.1.2.

Case 11.1.3. \( p_1 \neq @\text{nop} \), \( \text{break} \). Similar to Case 9.1.3.

Case 11.2. \( \text{isblocked}(p_1, n) \). Similar to Case 11.1. \( \square \)

**Theorem 3.14.** If \( \delta \overset{\pi}{\rightarrow} \delta' \) then \( \text{rank}(\delta) \geq \text{rank}(\delta') \).

**Proof.** If \( \delta \overset{\pi}{\rightarrow} \delta' \) then \( \delta \overset{\pi}{\rightarrow} \delta' \), for some \( i \). We proceed by induction on \( i \). The theorem is trivially true for \( i = 0 \) and follows directly from Lemma 3.13 for \( i = 1 \). Suppose \( \delta \overset{\pi}{\rightarrow} \delta' \), \( \delta' \overset{\pi}{\rightarrow} \delta' \), for some \( i > 1 \) and \( \delta' \). Thus, by Lemma 3.13 and by the induction hypothesis,
\[
\text{rank}(\delta) \geq \text{rank}(\delta'_1) \geq \text{rank}(\delta'). \quad \square
Theorem 3.15 (Termination). For any $\delta$, there is a $\delta' \in \Sigma^*$ such that $\delta \xrightarrow{\text{out}} \delta'$.

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

Basis. If $(i, j) = (0, 0)$ then $\delta$ cannot be advanced by $\xrightarrow{\text{out}}$, as $j = 0$ implies $e = \varepsilon$ and $n = 0$ (neither push nor pop can be applied). There are two possibilities: either $\delta$ is nested irreducible or it is not. In the first case, the theorem is trivially true, as $\xrightarrow{\text{out}} \delta_{\text{nst}}$. Suppose $\delta$ is not nested irreducible. Then, by Theorem 3.8, $\delta \xrightarrow{\text{out}} \delta'_{\text{nst}}$, for some $\delta'_{\text{nst}}$. By Theorem 3.14, $(i, j) = (0, 0) \geq \text{rank}(\delta')$.

which implies $\text{rank}(\delta') = (0, 0)$.

Induction. Let $(i, j) \neq (0, 0)$. There are two subcases.

Case 1. $\delta$ is nested-irreducible. There are two subcases.

Case 1.1. $j = 0$. By Definition 3.11, $\delta_{\text{st}}$. Thus $\delta \xrightarrow{\text{out}} \delta_{\text{st}}$.

Case 1.2. $j > 0$. There are two subcases. event.

Case 1.2.1. $e \neq e'$. Then, by push and by Theorem 3.8, there are $\delta_{\text{st}}^1$ and $\delta_{\text{nst}}^1 = (p', n + 1, e')$ such that

$$\delta \xrightarrow{\text{out}} \delta_{\text{st}}^1 \xrightarrow{\text{out}} \delta_{\text{nst}}^1.$$  

Thus, by item (a) of Lemma 3.12 and by Theorem 3.14,

$$\text{rank}(\delta) = \text{rank}(\delta_{\text{st}}^1) = (i, j) \geq \text{rank}(\delta') = (i', j') .$$

If $e' = e$, then $i = i'$ and $j = j'$, and the rest of this proof is similar to that of Case 1.2.2. Otherwise, if $e' \neq e$ then $i > i'$, since an emit $\text{out}(e')$ was consumed by the nested transitions. Thus,

$$\text{rank}(\delta) > \text{rank}(\delta') .$$

By the induction hypothesis, $\delta' \xrightarrow{\text{out}} \delta''$, for some $\delta''$. Therefore, $\delta \xrightarrow{\text{out}} \delta''$.

Case 1.2.2. $e = e'$. Then, since $j > 0$, $\delta \xrightarrow{\text{pop}} \delta'$, for some $\delta''$. By item (b) of Lemma 3.12,

$$\text{rank}(\delta) > \text{rank}(\delta') .$$

Hence, by the induction hypothesis, there is a $\delta''$ such that $\delta' \xrightarrow{\text{out}} \delta''$. Therefore, $\delta \xrightarrow{\text{out}} \delta''$.

Case 2. $\delta$ is not nested-irreducible. Then $e = e'$ and, by Theorems 3.8 and 3.14, there is a $\delta_{\text{nst}}^1$ such that $\delta \xrightarrow{\text{out}} \delta_{\text{nst}}$ with $\text{rank}(\delta) \geq \text{rank}(\delta_{\text{nst}}^1)$. The rest of this proof is similar to that of Case 1. \hfill $\Box$

B Artifact Appendix

B.1 Abstract

Our artifact includes an open-source implementation of the programming language Céu. The implementation is based on and should conform with the formal semantics presented in this paper. The artifact also includes an executable script with over 3500 test cases of valid and invalid programs in Céu. The script is customizable and allows to create new tests providing inputs and expected outputs. The evaluation platform is a Linux/Intel with Lua-5.3 and GCC-7.2 installed.

B.2 Artifact Check-List (Meta-Information)

- **Compilation**: The output of the Céu compiler is a C program that requires a C compiler (e.g., GCC-7.2).
- **Transformations**: The compiler of Céu is a Lua program (Lua-5.3) that generates a C program.
- **Data set**: The data set is a set of program test cases included in the language distribution.
- **Run-time environment**: Linux with Lua-5.3 and GCC-7.2.
- **Hardware**: An off-the-shelf Intel machine.
- **Execution**: 5-10 minutes for the full test.
- **Output**: Console output: Success (termination) or Fail (abortion).
- **Experiments**: Manual steps performed by the user in the command line.
- **Artifacts publicly available?**: Yes.
- **Artifacts functional?**: Yes.
- **Artifacts reusable?**: No.
- **Results validated?**: No.

B.3 Description

B.3.1 How Delivered

The compiler/distribution of Céu is available on GitHub: https://github.com/fsantanna/ceu

The website of Céu includes an introductory video and an online tutorial: http://www.ceu-lang.org/

The language also provides extensive documentation online: http://fsantanna.github.io/ceu/out/manual/v0.30/

B.3.2 Hardware Dependencies

An off-the-shelf Intel machine.

B.3.3 Software Dependencies

Linux, Lua-5.3 (with lpeg-1.0.0), and GCC-7.2.

B.3.4 Data Sets

A set of more than 3500 programs packed in a single script file which is included with the compiler distribution.

B.4 Installation

Install all required software (assuming an Ubuntu-based distribution):

```
$ sudo apt-get install git lua5.3 lua-lpeg liblua5.3-0 
1liblua5.3-dev
```

Clone the repository of Céu:

```
$ git clone https://github.com/fsantanna/ceu
$ cd ceu/
$ git checkout v0.30
Install Céu:
$ make
$ sudo make install # install as "/usr/local/bin/ceu"
```
B.5 Experiment Workflow

Run the experiment:

```
$ cd tst/
$ ./run.lua
```

B.6 Evaluation and Expected Result

The experiment will execute all test cases. In about 5-10 minutes, a summary will be printed on screen:

```
$ cd tst/
$ ./run.lua
<...> # output with the test programs
stats = {
  count = 3478,
  trails = 9082,
  bytes = 50803896,
  bcasts = 0,
  visits = 5189770,
}
```

B.7 Experiment Customization

B.7.1 Test Cases

The file `tst/tests.lua` includes all test cases. Each test case contains a program in Céu as well as the expected result, e.g.:

```
Test { [
  var int ret = 0;
  var int i;
  loop i in [0 -> 10] do
    ret = ret + 1;
  end
  escape ret;
],
  run = 10,
}
```

This test case should compile and run successfully yielding 10.

To customize the experiment, include a new test case at line 440, after the string "-- OK: well tested".

B.7.2 Sample Programs

The distribution of Céu comes with sample programs that can be executed as follows:

```
$ cd /<...>/ceu/ # change to the Céu repository
$ make samples # execute all samples one by one
```

B.7.3 Single File

It is also possible to compile a single file with a program in Céu:

```
$ cd /<...>/ceu/ # change to the Céu repository
$ make one CEU_SRC=/tmp/tst.ceu # compile a program
$ /tmp/tst # execute it
```

As an example, the file `/tmp/tst.ceu` may contain a program such as follows:

```
input int A; // two input events
input int B;
spawn do // generates test inputs asynchronously
  await async do
    emit A(10);
  end
end
var int a = await A; // main program
var int b = await B;
escape a+b; // 110 is the result of the program
```

Acknowledgments

This work was supported by the Serrapilheira Institute (grant number Serra-1708-15612).

References


