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

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

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

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

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

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

Abstract

Céu is a synchronous programming language for embedded soft real-time systems. It focuses on control-flow safety features, such as safe shared-memory concurrency and safe 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.

CCS Concepts • **Theory of computation → Operational semantics**; • **Software and its engineering → Concurrent programming languages**; • **Computer systems organization** → *Embedded software*;

Keywords Determinism, Termination, Operational semantics, Synchronous languages

ACM Reference Format:
Rodrigo C. M. Santos, Guilherme F. Lima, Francisco Sant'Anna, Roberto Ierusalimschy, and Edward H. Haeusler. 2018. A Memory-Bounded, Deterministic and Terminating Semantics for the Synchronous Programming Language Céu. In *Proceedings of 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3211332.3211334

LCTES'18, June 19–20, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5803-3/18/06…$15.00
https://doi.org/10.1145/3211332.3211334

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.

```
// 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 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.

```
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 radio receive, terminating on a button press always with the LED off.

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]

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.

[1] 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.

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`.

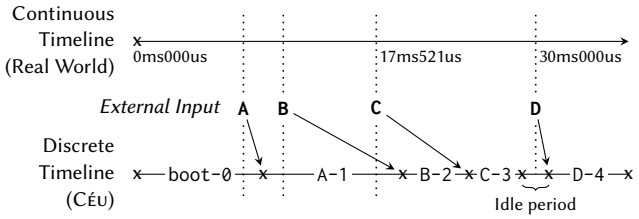

Figure 1. The discrete notion of time in Céu.

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.

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

```
1  event int* inc;          // declares subroutine "inc"
2  par/or do
3      var int* p;
4      every p in inc do // implements "inc" through 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.

(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).

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

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

[a] Accesses to `x` are never concurrent.

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

[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

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.

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

[a] Local resource finalization.

```
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
```

[b] External resource finalization.

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

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 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 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.

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

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 ::=	$\texttt{mem}(id)$	*any memory access to "id"*
	$\texttt{await}_{ext}(id)$	*await external event "id"*
	$\texttt{await}_{int}(id)$	*await internal event "id"*
	$\texttt{emit}_{int}(id)$	*emit internal event "id"*
	`break`	*loop escape*
	`if` $\texttt{mem}(id)$ `then` P_1 `else` P_2	*conditional*
	P_1 ; P_2	*sequence*
	`loop` P_1	*repetition*
	`every` id P_1	*event iteration*
	P_1 `and` P_2	*par/and*
	P_2 `or` P_2	*par/or*
	`fin` P	*finalization*
	P_1 `@loop` P_2	*unwinded loop*
	P_1 `@and` P_2	*unwinded par/and*
	P_1 `@or` P_2	*unwinded par/or*
	`@canrun`(n)	*can run on stack level n*
	`@nop`	*terminated program*

The $\texttt{mem}(id)$ primitive represents a read or write to the memory location identified by id.[3] 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 `fin`p 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 $\texttt{await}_{int}$ and $\texttt{emit}_{int}$ do not support such communication of values, it can be easily simulated: one can use a shared variable to hold the value of an $\texttt{emit}_{int}$ and access it after the corresponding $\texttt{await}_{int}$ 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).

[3]Although the same symbol id is used in their definition, `mem`, $\texttt{await}_{ext}$, $\texttt{await}_{int}$ and $\texttt{emit}_{int}$ 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 programs 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 a 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.,

$$\langle p, n, e\rangle \longrightarrow \langle p', n', e'\rangle\,,$$

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 ε).

We refer to the triples on the left-hand and right-hand sides of symbol $\longrightarrow$ as *descriptions* (denoted δ). The description on the left of $\longrightarrow$ 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[4]:

$$\langle p, 0, e\rangle \xrightarrow{*} \langle p', 0, \varepsilon\rangle\,.$$

We now proceed to give the rules for possible the transitions. We distinguish between two types of transitions: *outermost transitions* $\xrightarrow[out]{}$ and *nested transitions* $\xrightarrow[nst]{}$.

Outermost transitions

The rules **push** and **pop** for $\xrightarrow[out]{}$ transitions are non-recursive definitions that apply to the program as a whole. These are the only rules that manipulate the stack level.

$$\frac{e \neq \varepsilon}{\langle p, n, e\rangle \xrightarrow[out]{} \langle bcast(p, e), n+1, \varepsilon\rangle} \quad \textbf{(push)}$$

$$\frac{n > 0 \qquad p = \texttt{@nop} \vee p = \texttt{break} \vee isblocked(p, n)}{\langle p, n, \varepsilon\rangle \xrightarrow[out]{} \langle p, n-1, \varepsilon\rangle} \quad \textbf{(pop)}$$

Rule **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 $\texttt{await}_{ext}$ or $\texttt{await}_{int}$ statements (see *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 ε). Rule **push** is the only rule that matches an emitted event and also immediately consumes it.

[4] We write $\xrightarrow{i}$ to mean i transitions in sequence, and we write $\xrightarrow{*}$ to mean a finite (possibly zero) number of transitions in sequence.

Rule **pop** decreases the stack level by one and can only be applied if the program is blocked (see *isblocked* in Figure 4) or terminated ($p = \texttt{@nop}$ or $p = \texttt{break}$). This condition ensures that an $\texttt{emit}_{int}$ 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 **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 **pop** will lead to a description with this same program at stack level 0.

Nested transitions

The $\xrightarrow[nst]{}$ rules are recursive definitions of the form

$$\langle p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p', n, e\rangle.$$

Nested transitions 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 current stack level before the nested reaction is complete.

A complete reaction consists of a series of transitions

$$\langle p, 0, e_{ext}\rangle \xrightarrow[out]{push} \langle p_1, 1, \varepsilon\rangle \left[\xrightarrow[nst]{*} \xrightarrow[out]{}\right]* \xrightarrow[nst]{*} \xrightarrow[out]{pop} \langle p', 0, \varepsilon\rangle\,,$$

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

The $\xrightarrow[nst]{}$ rules for atoms are defined as follows:

$$\langle \texttt{mem}(id), n, \varepsilon\rangle \xrightarrow[nst]{} \langle \texttt{@nop}, n, \varepsilon\rangle \quad \textbf{(mem)}$$

$$\langle \texttt{emit}_{int}(id), n, \varepsilon\rangle \xrightarrow[nst]{} \langle \texttt{@canrun}(n), n, id\rangle \quad \textbf{(emit-int)}$$

$$\langle \texttt{@canrun}(n), n, \varepsilon\rangle \xrightarrow[nst]{} \langle \texttt{@nop}, n, \varepsilon\rangle \quad \textbf{(can-run)}$$

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

The rules for conditionals and sequences are the following:

$$\frac{eval(\texttt{mem}(id))}{\langle \texttt{if mem}(id) \texttt{ then } p \texttt{ else } q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p, n, \varepsilon\rangle} \quad \textbf{(if-true)}$$

$$\frac{\neg\, eval(\texttt{mem}(id))}{\langle \texttt{if mem}(id) \texttt{ then } p \texttt{ else } q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle q, n, \varepsilon\rangle} \quad \textbf{(if-false)}$$

$$\frac{\langle p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p', n, e\rangle}{\langle p\,;\, q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p';\, q, n, e\rangle} \quad \textbf{(seq-adv)}$$

$$\langle \texttt{@nop};\, q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle q, n, \varepsilon\rangle \quad \textbf{(seq-nop)}$$

$$\langle \texttt{break};\, q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \texttt{break}, n, \varepsilon\rangle \quad \textbf{(seq-brk)}$$

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:

$$\langle \texttt{loop}\ p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p\ \texttt{@loop}\ p, n, \varepsilon\rangle \quad \textbf{(loop-expd)}$$

$$\frac{\langle q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle q', n, e\rangle}{\langle q\ \texttt{@loop}\ p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle q'\ \texttt{@loop}\ p, n, e\rangle} \quad \textbf{(loop-adv)}$$

$$\langle \texttt{@nop @loop}\ p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \texttt{loop}\ p, n, \varepsilon\rangle \quad \textbf{(loop-nop)}$$

$$\langle \texttt{break @loop}\ p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \texttt{@nop}, n, \varepsilon\rangle \quad \textbf{(loop-brk)}$$

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:

$$\langle p\ \texttt{and}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p\ \texttt{@and (@canrun}(n); q), n, \varepsilon\rangle \quad \textbf{(and-expd)}$$

$$\frac{\langle p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p', n, e\rangle}{\langle p\ \texttt{@and}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p'\ \texttt{@and}\ q, n, e\rangle} \quad \textbf{(and-adv1)}$$

$$\frac{\mathit{isblocked}(p, n) \qquad \langle q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle q', n, e\rangle}{\langle p\ \texttt{@and}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p\ \texttt{@and}\ q', n, e\rangle} \quad \textbf{(and-adv2)}$$

$$\langle p\ \texttt{or}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p\ \texttt{@or (@canrun}(n); q), n, \varepsilon\rangle \quad \textbf{(or-expd)}$$

$$\frac{\langle p, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p', n, e\rangle}{\langle p\ \texttt{@or}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p'\ \texttt{@or}\ q, n, e\rangle} \quad \textbf{(or-adv1)}$$

$$\frac{\mathit{isblocked}(p, n) \qquad \langle q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle q', n, e\rangle}{\langle p\ \texttt{@or}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p\ \texttt{@or}\ q', n, e\rangle} \quad \textbf{(or-adv2)}$$

Rules **and-expd** and **or-expd** insert a `@canrun`(n) at the beginning of the right branch. This ensures that any $\texttt{emit}_{int}$ 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 **and-adv2** and **or-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**).

$$\langle \texttt{@nop @and}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle q, n, \varepsilon\rangle \quad \textbf{(and-nop1)}$$

$$\frac{\mathit{isblocked}(p, n)}{\langle p\ \texttt{@and @nop}, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p, n, \varepsilon\rangle} \quad \textbf{(and-nop2)}$$

$$\langle \texttt{@nop @or}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \mathit{clear}(q), n, \varepsilon\rangle \quad \textbf{(or-nop1)}$$

$$\frac{\mathit{isblocked}(p, n)}{\langle p\ \texttt{@or @nop}, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \mathit{clear}(p), n, \varepsilon\rangle} \quad \textbf{(or-nop2)}$$

The *clear* function (see Figure 4) concatenates all active `fin` bodies of the branch being aborted, so that they execute before the composition rejoins.

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 point, its body will execute as a result of the *clear* call. The body of a `fin` statement always execute within a reaction. This is due to a syntactic restriction: `fin` bodies cannot contain awaiting statements (namely, $\texttt{await}_{ext}$, $\texttt{await}_{int}$, `every`, or `fin`).

Finally, a `break` in one branch of a parallel escapes the closest enclosing `loop`, properly aborting the other branch with the *clear* function:

$$\langle \texttt{break @and}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \mathit{clear}(q); \texttt{break}, n, \varepsilon\rangle \quad \textbf{(and-brk1)}$$

$$\frac{\mathit{isblocked}(p, n)}{\langle p\ \texttt{@and break}, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \mathit{clear}(p); \texttt{break}, n, \varepsilon\rangle} \quad \textbf{(and-brk2)}$$

$$\langle \texttt{break @or}\ q, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \mathit{clear}(q); \texttt{break}, n, \varepsilon\rangle \quad \textbf{(or-brk1)}$$

$$\frac{\mathit{isblocked}(p, n)}{\langle p\ \texttt{@or break}, n, \varepsilon\rangle \xrightarrow[nst]{} \langle \mathit{clear}(p); \texttt{break}, n, \varepsilon\rangle} \quad \textbf{(or-brk2)}$$

A reaction eventually blocks in $\texttt{await}_{ext}$, $\texttt{await}_{int}$, `every`, `fin`, and `@canrun` statements in parallel trails. Then, if none of the trails is blocked in `@canrun`, it means that the program cannot advance in the current reaction. However, `@canrun` statements can still resume at lower stack indexes and will eventually resume in the current reaction (see rule **pop**).

3.3 Properties

3.3.1 Determinism

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 3.1 follows from a simple inspection of rules **push** and **pop**. The proof of Lemma 3.2 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 3.3.

(i) Function *bcast*:

$$
\begin{aligned}
bcast(\texttt{await}_{ext}(e), e) &= \texttt{@nop} \\
bcast(\texttt{await}_{int}(e), e) &= \texttt{@nop} \\
bcast(\texttt{every}\ e\ p, e) &= p;\ \texttt{every}\ e\ p \\
bcast(\texttt{@canrun}(n), e) &= \texttt{@canrun}(n) \\
bcast(\texttt{fin}\ p, e) &= \texttt{fin}\ p \\
bcast(p;\ q, e) &= bcast(p, e);\ q \\
bcast(p\ \texttt{@loop}\ q, e) &= bcast(p, e)\ \texttt{@loop}\ q \\
bcast(p\ \texttt{@and}\ q, e) &= bcast(p, e)\ \texttt{@and}\ bcast(q, e) \\
bcast(p\ \texttt{@or}\ q, e) &= bcast(p, e)\ \texttt{@or}\ bcast(q, e) \\
bcast(\_, e) &= \_\ \ (\texttt{mem}, \texttt{emit}_{int}, \texttt{break}, \\
&\qquad \texttt{if}\ \texttt{then}\ \texttt{else}, \texttt{loop}, \texttt{and}, \texttt{or}, \texttt{@nop})
\end{aligned}
$$

(ii) Predicate *isblocked*:

$$
\begin{aligned}
isblocked(\texttt{await}_{ext}(e), n) &= true \\
isblocked(\texttt{await}_{int}(e), n) &= true \\
isblocked(\texttt{every}\ e\ p, n) &= true \\
isblocked(\texttt{@canrun}(m), n) &= (n > m) \\
isblocked(\texttt{fin}\ p, n) &= true \\
isblocked(p;\ q, n) &= isblocked(p, n) \\
isblocked(p\ \texttt{@loop}\ q, n) &= isblocked(p, n) \\
isblocked(p\ \texttt{@and}\ q, n) &= isblocked(p, n) \wedge isblocked(q, n) \\
isblocked(p\ \texttt{@or}\ q, n) &= isblocked(p, n) \wedge isblocked(q, n) \\
isblocked(\_, n) &= false\ \ (\texttt{mem}, \texttt{emit}_{int}, \texttt{break}, \\
&\qquad \texttt{if}\ \texttt{then}\ \texttt{else}, \texttt{loop}, \texttt{and}, \texttt{or}, \texttt{@nop})
\end{aligned}
$$

(iii) Function *clear*:

$$
\begin{aligned}
clear(\texttt{await}_{ext}(e)) &= \texttt{@nop} \\
clear(\texttt{await}_{int}(e)) &= \texttt{@nop} \\
clear(\texttt{every}\ e\ p) &= \texttt{@nop} \\
clear(\texttt{@canrun}(n)) &= \texttt{@nop} \\
clear(\texttt{fin}\ p) &= p \\
clear(p;\ q) &= clear(p) \\
clear(p\ \texttt{@loop}\ q) &= clear(p) \\
clear(p\ \texttt{@and}\ q) &= clear(p);\ clear(q) \\
clear(p\ \texttt{@or}\ q) &= clear(p);\ clear(q) \\
clear(\_) &= \xi\ \ (\texttt{mem}, \texttt{emit}_{int}, \texttt{break}, \\
&\qquad \texttt{if}\ \texttt{then}\ \texttt{else}, \texttt{loop}, \texttt{and}, \texttt{or}, \texttt{@nop})
\end{aligned}
$$

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

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

Lemma 3.2. *If* $\delta \xrightarrow[nst]{} \delta_1$ *and* $\delta \xrightarrow[nst]{} \delta_2$ *then* $\delta_1 = \delta_2$.

The main result of this section, Theorem 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.

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

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

for some $i > 1$, δ_1' and δ_2'. Then, by Lemma 3.1 or 3.2, 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, $\delta_1 = \delta_2$. □

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 $\xrightarrow[nst]{}$ and $\xrightarrow[out]{}$.

Definition 3.4. A description $\delta = \langle p, n, e\rangle$ is *nested-irreducible* iff $e \neq \varepsilon$ or $p = \texttt{@nop}$ or $p = \texttt{break}$ or $isblocked(p, n)$.[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 3.4 is justified by Proposition 3.5, 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 3.5 follows directly from the definition of $\xrightarrow[nst]{}$ by contradiction on the hypothesis that there is such $k \neq i$.

Proposition 3.5. *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 3.6.
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) $\langle p_1\ \texttt{@and}\ p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_1'\ \texttt{@and}\ p_2, n, e'\rangle$.
(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) $\langle p_1\ \texttt{@or}\ p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_1\ \texttt{@or}\ p_2', n, e'\rangle$.

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

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 $p = \texttt{fin}\,p_1$ 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 $p_1' = \texttt{break}\,\texttt{@loop}\,p_1$ or $isblocked(p_1', n)$.

Theorem 3.8 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 3.6 and Assumption 3.7.

Theorem 3.8. *For any δ there is a $\delta'_{\#nst}$ such that $\delta \xrightarrow[nst]{*} \delta'_{\#nst}$.*

The main result of this section, Theorem 3.15, is similar to Theorem 3.8 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 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 $pot(p, e)$, 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 3.10. 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 3.11. 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 3.12 follows directly from **push** and **pop** and from Definitions 3.9 and 3.10. The proof of Lemma 3.13, however, is by induction on the structure of $\xrightarrow[nst]{}$ derivations.

Lemma 3.12.

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

The next theorem is a generalization of Lemma 3.13 for $\xrightarrow[nst]{*}$. Its proof follows from the lemma by induction on i.

Theorem 3.14. *If $\delta \xrightarrow[nst]{*} \delta'$ then $rank(\delta) \geq rank(\delta')$.*

We now state and prove the main result of this section, Theorem 3.15, 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 3.15 (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 3.8, $\delta \xrightarrow[nst]{*} \delta'_{\#nst}$, for some $\delta'_{\#nst}$. By Theorem 3.14, $rank(\delta) \geq rank(\delta')$, 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 3.11, $\delta_\#$, 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 3.8, 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 3.12 and by Theorem 3.14,

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

If $e' = \varepsilon$, then $i = i'$ and $j = j'$, and the rest of this proof is similar to that of Case 1.2.2 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 3.12, $rank(\delta) > rank(\delta')$. 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 3.8 and 3.14, 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. □

3.3.3 Memory bound

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.

4 Related Work

Céu 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, 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 [21] and Statemate [11] also distinguish internal from external events. In the former, *"reactions to external and internal events (...) can be sensed only after completion of the step"*. In the latter, *"the receiving state (of the internal event) acts here as a function"*. Although the descriptions suggest a stack-based semantics, we are not aware of formalizations for these ideas for a deeper comparison with Céu.

Like Céu, 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 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.

5 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. 2002. Cooperative Task Management Without Manual Stack Management. In *Proceedings of ATEC'02*. USENIX Association, 289–302.

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

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages twelve years later. In *Proceedings of the IEEE*, Vol. 91. 64–83.

[4] Gérard Berry. 1993. Preemption in Concurrent Systems.. In *FSTTCS (LNCS)*, Vol. 761. Springer, 72–93.

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

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

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

[8] Frédéric Boussinot and Robert De Simone. 1991. The Esterel language. *Proc. IEEE* 79, 9 (Sep 1991), 1293–1304.

[9] Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. 2005. The Synchronous Hypothesis and Synchronous Languages. In *Embedded Systems Handbook*, R. Zurawski (Ed.).

[10] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of SenSys'06*. ACM, 29–42.

[11] David Harel and Amnon Naamad. 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 5, 4 (1996), 293–333.

[12] Marcin Karpinski and Vinny Cahill. 2007. High-Level Application Development is Realistic for Wireless Sensor Networks. In *Proceedings of SECON'07*. 610–619.

[13] Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. *Deprecating the observer pattern*. Technical Report.

[14] ORACLE. 2011. Java Thread Primitive Deprecation. http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html (accessed in Aug-2014). (2011).

[15] Gordon D. Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report 19. Computer Science Departement, Aarhus University, Aarhus, Denmark.

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

[17] Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, Silvana Rossetto, and Adriano Branco. 2017. The Design and Implementation of the Synchronous Language CÉU. *ACM Trans. Embed. Comput. Syst.* 16, 4, Article 98 (July 2017), 26 pages. https://doi.org/10.1145/3035544

[18] Francisco Sant'Anna, Noemi Rodriguez, and Roberto Ierusalimschy. 2015. Structured Synchronous Reactive Programming with Céu. In *Proceedings of Modularity'15*.

[19] Francisco Sant'Anna, Noemi Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. 2013. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM.

[20] Rodrigo Santos, Guilherme Lima, Francisco Sant'Anna, and Noemi Rodriguez. 2016. Céu-Media: Local Inter-Media Synchronization Using Céu. In *Proceedings of WebMedia'16*. ACM, New York, NY, USA, 143–150. https://doi.org/10.1145/2976796.2976856

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

[22] Reinhard von Hanxleden. 2009. SyncCharts in C: a proposal for light-weight, deterministic concurrency. In *Proceedings EMSOFT'09*. ACM, 225–234.

A Detailed Proofs

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

Proof. The lemma is vacuously true if δ cannot be advanced by $\xrightarrow[out]{}$ transitions. Suppose that is not the case and let $\delta = \langle p, n, e\rangle$, $\delta_1 = \langle p_1, n_1, e_1\rangle$ and $\delta_2 = \langle p_2, n_2, e_2\rangle$. Then, there are two possibilities.

Case 1. $e \neq \varepsilon$. Both transitions are applications of **push**. Hence $p_1 = p_2 = bcast(p, e)$, $n_1 = n_2 = n + 1$, and $e_1 = e_2 = \varepsilon$.

Case 2. $e = \varepsilon$. Both transitions are applications of **pop**. Hence $p_1 = p_2 = p$, $n_1 = n_2 = n - 1$, and $e_1 = e_2 = \varepsilon$. □

Lemma 3.2. *If $\delta \xrightarrow[nst]{} \delta_1$ and $\delta \xrightarrow[nst]{} \delta_2$ then $\delta_1 = \delta_2$.*

Proof. By induction on the structure of $\xrightarrow[nst]{}$ derivations. The lemma is vacuously true if δ cannot be advanced by $\xrightarrow[nst]{}$ transitions. Suppose that is not the case and let $\delta = \langle p, n, e\rangle$, $\delta_1 = \langle p_1, n_1, e_1\rangle$ and $\delta_2 = \langle p_2, n_2, e_2\rangle$. Then, by the hypothesis of the lemma, there are derivations π_1 and π_2 such that

$$\pi_1 \Vdash \langle p, n, e\rangle \xrightarrow[nst]{} \langle p_1, n_1, e_1\rangle$$
$$\pi_2 \Vdash \langle p, n, e\rangle \xrightarrow[nst]{} \langle p_2, n_2, e_2\rangle$$

i.e., the conclusion of π_1 is $\langle p, n, e\rangle \xrightarrow[nst]{} \langle p_1, n_1, e_1\rangle$ and the conclusion of π_2 is $\langle p, n, e\rangle \xrightarrow[nst]{} \langle p_2, n_2, e_2\rangle$.

By definition of $\xrightarrow[nst]{}$, we have that $e = \varepsilon$ and $n_1 = n_2 = n$. It remains to be shown that $p_1 = p_2$ and $e_1 = e_2$.

Depending on the structure of program p, the following 11 cases are possible. (Note that p cannot be an $\texttt{await}_{ext}$, $\texttt{await}_{int}$, `break`, `every`, `fin`, or `@nop` statement as there is no $\xrightarrow[nst]{}$ rule to transition such programs.)

Case 1. $p = \texttt{mem}(id)$. Then derivations π_1 and π_2 are instances of rule **mem**, i.e., their conclusions are obtained by an application of this rule. Hence $p_1 = p_2 = \texttt{@nop}$ and $e_1 = e_2 = \varepsilon$.

Case 2. $p = \texttt{emit}_{int}(e')$. Then π_1 and π_2 are instances of **emit-int**. Hence $p_1 = p_2 = \texttt{@canrun}(n)$ and $e_1 = e_2 = e'$.

Case 3. $p = \texttt{@canrun}(n)$. Then π_1 and π_2 are instances of **canrun**. Hence $p_1 = p_2 = \texttt{@nop}$ and $e_1 = e_2 = \varepsilon$.

Case 4. $p = \texttt{if}\,\texttt{mem}(id)\,\texttt{then}\,p'\,\texttt{else}\,p''$. There are two subcases.

Case 4.1. $eval(\texttt{mem}(id))$. Then π_1 and π_2 are instances of **if-true**. Hence $p_1 = p_2 = p'$ and $e_1 = e_2 = \varepsilon$.

Case 4.2. $\neg\, eval(\texttt{mem}(id))$. Then π_1 and π_2 are instances of **if-false**. Hence $p_1 = p_2 = p''$ and $e_1 = e_2 = \varepsilon$.

Case 5. $p = p'; p''$. There are three subcases.

Case 5.1. $p' = \texttt{@nop}$. Then π_1 and π_2 are instances of **seq-nop**. Hence $p_1 = p_2 = p''$ and $e_1 = e_2 = \varepsilon$.

Case 5.2. $p' = \texttt{break}$. Then π_1 and π_2 are instances of **seq-brk**. Hence $p_1 = p_2 = \texttt{break}$ and $e_1 = e_2 = \varepsilon$.

Case 5.3. $p' \neq \texttt{@nop}, \texttt{break}$. Then π_1 and π_2 are instances of **seq-adv**. Thus there are derivations π_1' and π_2' such that

$$\pi_1' \Vdash \langle p', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_1', n, e_1'\rangle$$
$$\pi_2' \Vdash \langle p', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_2', n, e_2'\rangle$$

for some p_1', p_2', e_1', and e_2'. By the induction hypothesis, $p_1' = p_2'$ and $e_1' = e_2'$. Hence $p_1 = p_1'; p'' = p_2'; p'' = p_2$ and $e_1 = e_1' = e_2' = e_2$.

Case 6. $p = \texttt{loop}\,p'$. Then π_1 and π_2 are instances of **loop-expd**. Hence $p_1 = p_2 = p'\,\texttt{@loop}\,p'$ and $e_1 = e_2 = \varepsilon$.

Case 7. $p = p'\,\texttt{@loop}\,p''$. There are three subcases.

Case 7.1. $p' = \texttt{@nop}$. Then π_1 and π_2 are instances of **loop-nop**. Hence $p_1 = p_2 = \texttt{loop}\,p''$ and $e_1 = e_2 = \varepsilon$.

Case 7.2. $p' = \texttt{break}$. Then π_1 and π_2 are instances of **loop-brk**. Hence $p_1 = p_2 = \texttt{@nop}$ and $e_1 = e_2 = \varepsilon$.

Case 7.3. $p' \neq \texttt{@nop}, \texttt{break}$. Then π_1 and π_2 are instances of **loop-adv**. Thus there are derivations π_1' and π_2' such that

$$\pi_1' \Vdash \langle p', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_1', n, e_1'\rangle$$
$$\pi_2' \Vdash \langle p', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_2', n, e_2'\rangle$$

for some p_1', p_2', e_1', and e_2'. By the induction hypothesis, $p_1' = p_2'$ and $e_1' = e_2'$. Hence $p_1 = p_1'\,\texttt{@loop}\,p'' = p_2'\,\texttt{@loop}\,p'' = p_2$ and $e_1 = e_1' = e_2' = e_2$.

Case 8. $p = p'$ and p''. Then π_1 and π_2 are instances of **and-expd**. Hence $p_1 = p_2 = p'\,\texttt{@and}\,(\texttt{@canrun}(n);\, p'')$ and $e_1 = e_2 = \varepsilon$.

Case 9. $p = p'\,\texttt{@and}\,p''$. There are two subcases.

Case 9.1. $\neg\, isblocked(p', n)$. There are three subcases.

Case 9.1.1. $p' = \texttt{@nop}$. Then π_1 and π_2 are instances of **and-nop1**. Hence $p_1 = p_2 = p''$ and $e_1 = e_2 = \varepsilon$.

Case 9.1.2. $p' = \texttt{break}$. Then π_1 and π_2 are instances of **and-brk1**. Hence $p_1 = p_2 = clear(p'');\texttt{break}$ and $e_1 = e_2 = \varepsilon$.

Case 9.1.3. $p' \neq \texttt{@nop}, \texttt{break}$. Then π_1 and π_2 are instances of **and-adv1**. Thus there are derivations π_1' and π_2' such that

$$\pi_1' \Vdash \langle p', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_1', n, e_1'\rangle$$
$$\pi_2' \Vdash \langle p', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_2', n, e_2'\rangle$$

for some p_1', p_2', e_1', e_2'. By the induction hypothesis, $p_1' = p_2'$ and $e_1' = e_2'$. Hence $p_1 = p_1'$ and $p'' = p_2'$ and $p'' = p_2$ and $e_1 = e_1' = e_2' = e_2$.

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

Case 9.2.1. $p'' = \texttt{@nop}$. Then π_1 and π_2 are instances of **and-nop2**. Hence $p_1 = p_2 = p'$ and $e_1 = e_2 = \varepsilon$.

Case 9.2.2. $p'' = \texttt{break}$. Then π_1 and π_2 are instances of **and-brk2**. Hence $p_1 = p_2 = clear(p');\texttt{break}$ and $e_1 = e_2 = \varepsilon$.

Case 9.2.3. $p'' \neq$ `@nop, break`. Then π_1 and π_2 are instances of **and-adv2**. Thus there are derivations π_1'' and π_2'' such that

$$\pi_1'' \Vdash \langle p'', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_1'', n, e_1''\rangle$$
$$\pi_2'' \Vdash \langle p'', n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_2'', n, e_2''\rangle$$

for some p_1'', p_2'', e_1'', and e_2''. By the induction hypothesis, $p_1'' = p_2''$ and $e_1'' = e_2''$. Hence $p_1 = p'$ and $p_1'' = p'$ and $p_2'' = p_2$ and $e_1 = e_1'' = e_2'' = e_2$.

Case 10. $p = p'$ `or` p''. Then π_1 and π_2 are instances of **or-expd**. Hence $p_1 = p_2 = p'$ `@or` (`@canrun`(n); p'') and $e_1 = e_2 = \varepsilon$.

Case 11. $p = p'$ `@or` p''. There are two subcases.

Case 11.1. $\neg$ *isblocked*(p', n). There are three subcases.

Case 11.1.1. $p' =$ `@nop`. Then π_1 and π_2 are instances of **or-nop1**. Hence $p_1 = p_2 = clear(p'')$ and $e_1 = e_2 = \varepsilon$.

Case 11.1.2. $p' =$ `break`. Similar to Case 9.1.2.

Case 11.1.3. $p' \neq$ `@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 π_1 and π_2 are instances of **or-nop1**. Hence $p_1 = p_2 = clear(p')$ and $e_1 = e_2 = \varepsilon$.

Case 11.2.2. $p'' =$ `break`. Similar to Case 9.2.2.

Case 11.2.3. $p'' \neq$ `@nop, break`. Similar to Case 9.2.3.

□

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 Lemmas 3.1 and 3.2 for $i = 1$. 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'. There are two possibilities.

Case 1. $\delta \xrightarrow[out]{1} \delta_1'$ and $\delta \xrightarrow[out]{1} \delta_2'$. Then, by Lemma 3.1, $\delta_1' = \delta_2'$, and by the induction hypothesis, $\delta_1 = \delta_2$.

Case 2. $\delta \xrightarrow[nst]{1} \delta_1'$ and $\delta \xrightarrow[nst]{1} \delta_2'$. Then, by Lemma 3.2, $\delta_1' = \delta_2'$, and by the induction hypothesis, $\delta_1 = \delta_2$. □

Proposition 3.5. *If $\delta \xrightarrow[nst]{i} \delta'_{\#nst}$ then, for all $k \neq i$, there is no $\delta''_{\#nst}$ such that $\delta \xrightarrow[nst]{k} \delta''_{\#nst}$.*

Proof. By contradiction on the hypothesis that there is such k. Let $\delta \xrightarrow[nst]{i} \delta'_{\#nst}$, for some $i \geq 0$. There are two cases.

Case 1. Suppose there are $k > i$ and $\delta''_{\#nst}$ such that $\delta \xrightarrow[nst]{k} \delta''$. Then, by definition of $\xrightarrow[nst]{k}$,

$$\delta \xrightarrow[nst]{i} \delta' \xrightarrow[nst]{i+1} \delta_1' \xrightarrow[nst]{i+2} \cdots \xrightarrow[nst]{k} \delta''. \tag{1}$$

Since $\delta' = \langle p', n, e'\rangle$ is nested-irreducible, $e' = \varepsilon$ or $p =$ `@nop, break` or *isblocked*(p', n). In any of these cases, by the definition of $\xrightarrow[nst]{}$, there is no δ_1' such that $\delta' \xrightarrow[nst]{1} \delta_1'$, which contradicts (1). Therefore, no such k can exist.

Case 2. Suppose there are $k < i$ and $\delta''_{\#nst}$ such that $\delta \xrightarrow[nst]{k} \delta''$. Then, since $i > k$, by Case 1, δ' cannot exist, which is absurd. Therefore, the assumption that there is such k is false. □

Lemma 3.6.

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

Proof. By induction on i.

(a) The lemma is trivially true for $i = 0$, as $p_1 = p_1'$, and follows directly from **seq-adv** for $i = 1$. Suppose

$$\langle p_1, n, e\rangle \xrightarrow[nst]{1} \langle p_1'', n, e''\rangle \xrightarrow[nst]{i-1} \langle p_1', n, e'\rangle\,, \tag{2}$$

for some $i > 1$. Then $\langle p_1'', n, e''\rangle$ is not nested-irreducible, i.e., $e = \varepsilon$ and $p \neq$ `@nop, break` and $\neg$ *isblocked*(p_1'', n). By (2) and by **seq-adv**,

$$\langle p_1;\, p_2, n, e\rangle \xrightarrow[nst]{1} \langle p_1'';\, p_2, n, e''\rangle\,. \tag{3}$$

From (2), by the induction hypothesis,

$$\langle p_1'';\, p_2, n, e''\rangle \xrightarrow[nst]{i-1} \langle p_1';\, p_2, n, e'\rangle\,. \tag{4}$$

From (3) and (4),

$$\langle p_1;\, p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_1';\, p_2, n, e'\rangle\,.$$

(b) Similar to item (a).
(c) Similar to item (a).
(d) Similar to item (a).
(e) The lemma is trivially true for $i = 0$, as $p_2 = p_2'$, and follows directly from **and-adv2** for $i = 1$. Suppose

$$\langle p_2, n, e\rangle \xrightarrow[nst]{1} \langle p_2'', n, e''\rangle \xrightarrow[nst]{i-1} \langle p_2', n, e'\rangle\,, \tag{5}$$

for some $i > 1$. Then $\langle p_2'', n, e''\rangle$ is not nested-irreducible. By (5) and by **and-adv2**,

$$\langle p_1$ `@and` $p_2, n, e\rangle \xrightarrow[nst]{1} \langle p_1$ `@and` $p_2'', n, e''\rangle\,. \tag{6}$$

From (5), by the induction hypothesis,

$$\langle p_1$ `@and` $p_2'', n, e''\rangle \xrightarrow[nst]{i-1} \langle p_1$ `@or` $p_2', n, e'\rangle\,. \tag{7}$$

From (6) and (7),

$$\langle p_1$ `@and` $p_2, n, e\rangle \xrightarrow[nst]{i} \langle p_1$ `@and` $p_2', n, e'\rangle\,.$$

(f) Similar to item (a). □

Theorem 3.8. *For any δ there is a $\delta'_{\#nst}$ such that $\delta \xrightarrow[nst]{*} \delta'_{\#nst}$.*

Proof. By induction on the structure of programs. Let $\delta = \langle p, n, \varepsilon\rangle$. The theorem is trivially true if δ is nested-irreducible, as by definition $\delta \xrightarrow[nst]{0} \delta$. Suppose that is not the case. Then, depending on the structure of p, there are 11 possibilities. In each one of them, we show that such $\delta'_{\#nst}$ indeed exists.

Case 1. $p = \texttt{mem}(id)$. Then, by **mem**,

$$\langle \texttt{mem}(id), n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \texttt{@nop}, n, \varepsilon\rangle_{\#nst}\,.$$

Case 2. $p = \texttt{emit}_{int}(e)$. Then, by **emit-int**,

$$\langle \texttt{emit}_{int}(e), n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \texttt{@canrun}(n), n, e\rangle_{\#nst}\,.$$

Case 3. $p = \texttt{@canrun}(n)$. Then, by **can-run**,

$$\langle \texttt{@canrun}(n), n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \texttt{@nop}, n, \varepsilon\rangle_{\#nst}\,.$$

Case 4. $p = \texttt{if mem}(id)\ \texttt{then}\ p'\ \texttt{else}\ p''$. There are two subcases.

Case 4.1. *eval*($\texttt{mem}(id)$). Then, by **if-true** and by the induction hypothesis, there is a δ' such that

$$\langle \texttt{if mem}(id)\ \texttt{then}\ p'\ \texttt{else}\ p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle p', n, e\rangle \xrightarrow[nst]{*} \delta'_{\#nst}\,.$$

Case 4.2. $\neg$ *eval*($\texttt{mem}(id)$). Similar to Case 4.1.

Case 5. $p = p'; p''$. There are three subcases.

Case 5.1. $p' = \texttt{@nop}$. Then, by **seq-nop** and by the induction hypothesis, there is a δ' such that

$$\langle \texttt{@nop}; p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle p'', n, e\rangle \xrightarrow[nst]{*} \delta'_{\#nst}\,.$$

Case 5.2. $p' = \texttt{break}$. Then, by **seq-brk**,

$$\langle \texttt{break}; p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \texttt{break}, n, \varepsilon\rangle_{\#nst}\,.$$

Case 5.3. $p' \neq \texttt{@nop}, \texttt{break}$. Then, by the induction hypothesis, there are p'_1 and e such that

$$\langle p', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p'_1, n, e\rangle_{\#nst}\,.$$

By item (a) of Lemma 3.6,

$$\langle p'; p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p'_1; p'', n, e\rangle\,. \tag{8}$$

It remains to be shown that $\langle p'_1; p'', n, e\rangle$ is nested-irreducible. There are four possibilities following from the fact that the simpler $\langle p'_1, n, e\rangle$ is nested-irreducible.

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

Case 5.3.2. $p'_1 = \texttt{@nop}$. From (8),

$$\langle p'; p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle \texttt{@nop}; p'', n, e\rangle\,.$$

From this point on, this case is similar to Case 5.1.

Case 5.3.3. $p'_1 = \texttt{break}$. From (8),

$$\langle p'; p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle \texttt{break}; p'', n, e\rangle\,.$$

From this point on, this case is similar to Case 5.2.

Case 5.3.4. *isblocked*(p'_1, n). Then, by definition,

$$\mathit{isblocked}(p'_1; p'', n) = \mathit{isblocked}(p'_1, n) = \mathit{true}\,.$$

Hence, from (8) and by the definition #*nst*, description $\langle p'_1; p'', n, e\rangle$ is nested-irreducible.

Case 6. $p = \texttt{loop}\,p'$. Then, by item (b) of Assumption 3.7,

$$\langle \texttt{loop}\,p', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p'_1, n, e\rangle\,, \tag{9}$$

for some e and p'_1 such that either $p'_1 = \texttt{break @loop}\,p'$ or *isblocked*(p'_1, n).

Case 6.1. $p'_1 = \texttt{break @loop}\,p'$. From (9), by **loop-brk**,

$$\langle \texttt{loop}\,p', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle \texttt{break @loop}\,p', n, e\rangle \xrightarrow[nst]{1} \langle \texttt{@nop}, n, e\rangle_{\#nst}\,.$$

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

Case 7. $p = p'\ \texttt{@loop}\,p''$. There are three subcases.

Case 7.1. $p' = \texttt{@nop}$. Then, by **loop-nop**,

$$\langle \texttt{@nop @loop}\,p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \texttt{loop}\,p'', n, \varepsilon\rangle\,.$$

From this point on, this case is similar to Case 6.

Case 7.2. $p' = \texttt{break}$. Then, by **loop-brk**,

$$\langle \texttt{break @loop}\,p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \texttt{@nop}, n, \varepsilon\rangle_{\#nst}\,.$$

Case 7.3. $p' \neq \texttt{@nop}, \texttt{break}$. Then, by the induction hypothesis, there are p'_1 and e such that

$$\langle p', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p'_1, n, e\rangle_{\#nst}\,.$$

By item (b) of Lemma 3.6,

$$\langle p'\ \texttt{@loop}\,p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p'_1\ \texttt{@loop}\,p'', n, e\rangle\,.$$

It remains to be show that $\langle p'_1\ \texttt{@loop}\,p'', n, e\rangle$ is nested-irreducible. The rest of this proof is similar to that of Case 5.3.

Case 8. $p = p'\ \texttt{and}\ p''$. Then, by **and-expd**,

$$\langle p'\ \texttt{and}\ p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle p'\ \texttt{@and}\ (\texttt{@canrun}(n); p''), n, \varepsilon\rangle\,.$$

From this point on, this case is similar to Case 9.

Case 9. $p = p'\ \texttt{@and}\ p''$. There are two subcases.

Case 9.1. $\neg$ *isblocked*(p', n). There are three subcases.

Case 9.1.1. $p' = \texttt{@nop}$. Then, by **and-nop1** and by the induction hypothesis, there is a δ' such that

$$\langle \texttt{@nop @and}\,p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \delta'_{\#nst}\,.$$

Case 9.1.2. $p' = \texttt{break}$. Then, by **and-brk1**,

$$\langle \texttt{break @and}\,p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \mathit{clear}(p''); \texttt{break}, n, \varepsilon\rangle\,. \tag{10}$$

From (10), by item (a) of Assumption 3.7 and by **seq-nop**,

$$\langle \mathit{clear}(p''); \texttt{break}, n, \varepsilon\rangle \xrightarrow[nst]{*} \langle \texttt{@nop}; \texttt{break}, n, \varepsilon\rangle \xrightarrow[nst]{1} \langle \texttt{break}, n, \varepsilon\rangle_{\#nst}\,.$$

Case 9.1.3. $p' \neq \texttt{@nop}, \texttt{break}$. Then, by the induction hypothesis, there are p'_1 and e such that

$$\langle p', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p'_1, n, e\rangle_{\#nst}\,.$$

By item (c) of Lemma 3.6,

$$\langle p'\ \texttt{@and}\ p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p'_1\ \texttt{@and}\ p'', n, e\rangle\,.$$

It remains to be show that $\langle p_1' \texttt{ @and } p'', n, e\rangle$ 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 $e \neq \varepsilon$ then, by definition, $\langle p_1' \texttt{ @and } p'', n, e\rangle_{\#nst}$.
2. If $p_1' = \texttt{@nop}$, this case is similar to Case 9.1.1.
3. If $p_1' = \texttt{break}$, this case is similar to Case 9.1.2.
4. If $isblocked(p_1', n)$, this case is similar to Case 9.2.

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

Case 9.2.1. $p'' = \texttt{@nop}$. Then, by **and-nop2**,

$$\langle p' \texttt{ @and @nop}, n, \varepsilon\rangle \xrightarrow[nst]{1} \langle p', n, \varepsilon\rangle_{\#nst} .$$

Case 9.2.2. $p'' = \texttt{break}$. Then, by **and-brk2**,

$$\langle p' \texttt{ @and break}, n, \varepsilon\rangle \xrightarrow[nst]{1} \langle clear(p'); \texttt{break}, n, \varepsilon\rangle .$$

From this point on, this case is similar to Case 9.1.2.

Case 9.2.3. $p'' \neq \texttt{@nop}, \texttt{break}$. Then, by the induction hypothesis, there are p_1'' and e such that

$$\langle p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p_1'', n, e\rangle_{\#nst} .$$

By item (a) of Lemma 3.6,

$$\langle p' \texttt{ @and } p'', n, \varepsilon\rangle \xrightarrow[nst]{*} \langle p' \texttt{ @and } p_1'', n, e\rangle .$$

It remains to be show that $\langle p' \texttt{ @and } p_1'', n, e\rangle$ 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 $e \neq \varepsilon$ then, by definition, $\langle p' \texttt{ @and } p_1'', n, e\rangle_{\#nst}$.
2. If $p_1'' = \texttt{@nop}$, this case is similar to Case 9.2.1.
3. If $p_1'' = \texttt{break}$, this case is similar to Case 9.2.2.
4. If $isblocked(p_1'', n)$ then, as both sides are blocked, by definition, $\langle p' \texttt{ @and } p_1'', n, e\rangle_{\#nst}$.

Case 10. $p = p' \texttt{ or } p''$. Then, by **or-expd**,

$$\langle p' \texttt{ or } p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle p' \texttt{ @or } (\texttt{@canrun}(n); p''), n, \varepsilon\rangle .$$

From this point on, this case is similar to Case 11.

Case 11. $p = p' \texttt{ @or } p''$. There are two subcases.

Case 11.1. $\neg\, isblocked(p', n)$. There are three subcases.

Case 11.1.1. $p' = \texttt{@nop}$. Then, by **or-nop1**,

$$\langle \texttt{@nop @or } p'', n, \varepsilon\rangle \xrightarrow[nst]{1} \langle clear(p''), n, \varepsilon\rangle . \tag{11}$$

From (11), by item (a) Assumption 3.7,

$$\langle clear(p''), n, \varepsilon\rangle \xrightarrow[nst]{*} \langle \texttt{@nop}, n, \varepsilon\rangle_{\#nst} .$$

Case 11.1.2. $p' = \texttt{break}$. Similar to Case 9.1.2.

Case 11.1.3. $p' \neq \texttt{@nop}, \texttt{break}$. Similar to Case 9.1.3.

Case 11.2. $isblocked(p', n)$. There are three subcases.

Case 11.2.1. $p'' = \texttt{@nop}$. Then, by **or-nop2**,

$$\langle p' \texttt{ @or @nop}, n, \varepsilon\rangle \xrightarrow[nst]{1} \langle clear(p'), n, \varepsilon\rangle . \tag{12}$$

From (12), by item (a) of Assumption 3.7 and by definition of *clear*,

$$\langle clear(p'), n, \varepsilon\rangle \xrightarrow[nst]{*} \langle \texttt{@nop}, n, \varepsilon\rangle_{\#nst} .$$

Case 11.2.2. $p'' = \texttt{break}$. Similar to Case 9.2.2.

Case 11.2.3. $p'' \neq \texttt{@nop}, \texttt{break}$. Similar to Case 9.2.3. □

Lemma 3.12.

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

Proof. Let $\delta = \langle p, n, e\rangle$, $\delta' = \langle p', n', e'\rangle$, $rank(\delta) = \langle i, j\rangle$, and $rank(\delta') = \langle i', j'\rangle$.

(a) Suppose $\langle p, n, e\rangle \xrightarrow[out]{push} \langle p', n', e'\rangle$. Then, by **push**, $e \neq \varepsilon$, $p' = bcast(p, e)$, $n' = n + 1$, and $e' = \varepsilon$. By Definition 3.10, $j = n + 1$, as $e \neq \varepsilon$, and $j' = n + 1$, as $e' = \varepsilon$ and $n' = n + 1$; hence $j = j'$. It remains to be shown that $i = i'$:

$$\begin{aligned}
i &= pot(p, e) && \text{by Definition 3.10}\\
&= pot'(bcast(p, e)) && \text{by Definition 3.9}\\
&= pot'(p') && \text{since } p' = bcast(p, e)\\
&= pot'(bcast(p', \varepsilon)) && \text{by definition of } bcast\\
&= pot'(bcast(p', e')) && \text{since } e' = \varepsilon\\
&= pot(p', e') && \text{by Definition 3.9}\\
&= i' && \text{by Definition 3.10}
\end{aligned}$$

Therefore, $\langle i, j\rangle = \langle i', j'\rangle$.

(b) Suppose $\langle p, n, e\rangle \xrightarrow[out]{pop} \langle p', n', e'\rangle$. Then, by **pop**, $p = p'$, $n > 0$, $n' = n - 1$, and $e = e' = \varepsilon$. By Definition 3.9, $pot(bcast(p, e)) = pot(bcast(p', e'))$; hence $i = i'$. And by Definition 3.10, $j = n$, as $e = \varepsilon$, and $j' = n - 1$, as $e' = \varepsilon$ and $n' = n - 1$; hence $j > j'$. Therefore, $\langle i, j\rangle > \langle i', j'\rangle$. □

Lemma 3.13. *If* $\delta \xrightarrow[nst]{} \delta'$ *then* $rank(\delta) \geq rank(\delta')$.

Proof. We proceed by induction on the structure of $\xrightarrow[nst]{}$ derivations. Let $\delta = \langle p, n, e\rangle$, $\delta' = \langle p', n', e'\rangle$, $rank(\delta) = \langle i, j\rangle$, and $rank(\delta') = \langle i', j'\rangle$. By the hypothesis of the lemma, there is a derivation π such that

$$\pi \Vdash \langle p, n, e\rangle \xrightarrow[nst]{} \langle p', n', e'\rangle .$$

By definition of $\xrightarrow[nst]{}$, $e = \varepsilon$ and $n = n'$. Depending on the structure of program p, there are 11 possibilities. In each one of them we show that $rank(\delta) \geq rank(\delta')$.

Case 1. $p = \texttt{mem}(id)$. Then π is an instance of **mem**. Hence $p' = \texttt{@nop}$ and $e' = \varepsilon$. Thus $rank(\delta) = rank(\delta') = \langle 0, n\rangle$.

Case 2. $p = \texttt{emit}_{int}(e_1)$. Then π is an instance of **emit-int**. Hence $p' = \texttt{@canrun}$ and $e' = e_1 \neq \varepsilon$. Thus

$$rank(\delta) = \langle 1, n\rangle > \langle 0, n + 1\rangle = rank(\delta') .$$

Case 3. $p = \texttt{@canrun}(n)$. Then π is an instance of **can-run**. Hence $p' = \texttt{@nop}$ and $e' = \varepsilon$. Thus

$$rank(\delta) = rank(\delta') = \langle 0, n\rangle\,.$$

Case 4. $p = \texttt{if}\ p\ \texttt{then}\ p_1\ \texttt{else}\ p_2$. There are two subcases.

Case 4.1. $eval(\texttt{mem}(id))$. Then π is an instance of **if-true**. Hence $p' = p_1$ and $e' = \varepsilon$. Thus

$$\begin{aligned} rank(\delta) &= \langle \max\{pot'(p_1), pot'(p_2)\}, n\rangle \\ &\geq \langle pot'(p_1), n\rangle = rank(\delta')\,. \end{aligned}$$

Case 4.2. $\neg\, eval(\texttt{mem}(id))$. Similar to Case 4.1.

Case 5. $p = p_1; p_2$. There are three subcases.

Case 5.1. $p_1 = \texttt{@nop}$. Then π is an instance of **seq-nop**. Hence $p' = p_2$ and $e' = \varepsilon$. Thus

$$\begin{aligned} rank(\delta) &= \langle pot'(p_1) + pot'(p_2), n\rangle \\ &\geq \langle pot'(p_2), n\rangle = rank(\delta')\,. \end{aligned}$$

Case 5.2. $p_1 = \texttt{break}$. Then π is an instance of **seq-brk**. Hence $p' = p_1$ and $e' = \varepsilon$. Thus

$$rank(\delta) = rank(\delta') = \langle 0, n\rangle\,.$$

Case 5.3. $p_1 \neq \texttt{@nop}, \texttt{break}$. Then π is an instance of **seq-adv**. Hence there is a derivation π' such that

$$\pi' \Vdash \langle p_1, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_1', n, e_1'\rangle\,,$$

for some p_1' and e_1'. Thus $p' = p_1'; p_2$ and $e' = e_1'$. By the induction hypothesis,

$$rank(\langle p_1, n, \varepsilon\rangle) \geq rank(\langle p_1', n, e_1'\rangle)\,. \tag{13}$$

There are two subcases.

Case 5.3.1. $e' = \varepsilon$ Then

$$\begin{aligned} rank(\delta) &= \langle pot'(p_1) + pot'(p_2), n\rangle \ \text{ and} \\ rank(\delta') &= \langle pot'(p_1') + pot'(p_2), n\rangle\,. \end{aligned}$$

By (13), $pot'(p_1) \geq pot'(p_1')$. Thus

$$rank(\delta) \geq rank(\delta')\,.$$

Case 5.3.2. $e' \neq \varepsilon$. Then π' contains one application of **emit-int**, which consumes one $\texttt{emit}_{int}(e')$ statement from p_1 and implies $pot'(p_1) > pot'(p_1')$. Thus

$$\begin{aligned} rank(\delta) &= \langle pot'(p_1) + pot'(p_2), n\rangle \\ &> \langle pot'(p_1') + pot'(p_2), n+1\rangle = rank(\delta')\,. \end{aligned}$$

Case 6. $p = \texttt{loop}\, p_1$. Then π is an instance of **loop-expd**. Hence $p' = p_1\ \texttt{@loop}\, p_1$ and $e' = \varepsilon$. By item (b) of Assumption 3.7, all execution paths of p_1 contain at least one occurrence of break or $\texttt{await}_{ext}$. Thus, by condition (†) in Definition 3.9,

$$rank(\delta) = rank(\delta') = \langle pot'(p_1), n\rangle\,.$$

Case 7. $p = p_1\ \texttt{@loop}\, p_2$. There are three subcases.

Case 7.1. $p_1 = \texttt{@nop}$. Similar to Case 5.1.

Case 7.2. $p_1 = \texttt{break}$. Similar to Case 5.2.

Case 7.3. $p_1 \neq \texttt{@nop}, \texttt{break}$. Then π is an instance of **loop-adv**. Hence there is a derivation π' such that

$$\pi' \Vdash \langle p_1, n, \varepsilon\rangle \xrightarrow[nst]{} \langle p_1', n, e_1'\rangle\,,$$

for some p_1' and e_1'. Thus $p' = p_1'\ \texttt{@loop}\ p_2$ and $e' = e_1'$. There are two subcases.

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

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

Case 8. $p = p_1\ \texttt{and}\ p_2$. Then π is an instance of **and-expd**. Hence $p' = p_1\ \texttt{@and}\ (\texttt{@canrun}(n);\ p_2)$ and $e' = \varepsilon$. Thus

$$rank(\delta) = rank(\delta') = \langle pot'(p_1) + pot'(p_2), n\rangle\,.$$

Case 9. $p = p_1\ \texttt{@and}\ p_2$. There are two subcases. stack level n.

Case 9.1. $\neg\, isblocked(p_1, n)$. There are three subcases.

Case 9.1.1. $p_1 = \texttt{@nop}$. Then π is an instance of **and-nop1**. Hence $p' = p_2$ and $e' = \varepsilon$. Thus

$$rank(\delta) = rank(\delta') = \langle 0 + pot'(p_2), n\rangle\,.$$

Case 9.1.2. $p_1 = \texttt{break}$. Then π is an instance of **and-brk1**. Hence $p' = clear(p_2); \texttt{break}$ and $e' = \varepsilon$. By item (a) of Assumption 3.7 and by the definition of *clear*, $clear(p_2)$ does not contain $\texttt{emit}_{int}$ statements. Thus

$$rank(\delta) = rank(\delta') = \langle 0, n\rangle\,.$$

Case 9.1.3. $p_1 \neq \texttt{@nop}, \texttt{break}$. Then π is an instance of **and-adv1**. As $p_1 \neq \texttt{break}$ and $p_2 \neq \texttt{break}$ (otherwise **and-brk2** would have taken precedence), the rest of this proof is similar to that of Case 5.3.

Case 9.2. $isblocked(p_1, n)$. Similar to Case 9.1

Case 10. $p = p_1\ \texttt{or}\ p_2$. Then π is an instance of **or-expd**. Hence $p' = p_1\ \texttt{@or}\ (\texttt{@canrun}(n);\ p_2)$ and $e' = \varepsilon$. Thus

$$rank(\delta) = rank(\delta') = \langle pot'(p_1) + pot'(p_2), n\rangle\,.$$

Case 11. $p = p_1\ \texttt{@or}\ p_2$. There are two subcases.

Case 11.1. $\neg\, isblocked(p_1, n)$. There are three subcases.

Case 11.1.1. $p_1 = \texttt{@nop}$. Then π is an instance of **or-nop1**. Hence $p' = clear(p_2)$ and $e' = \varepsilon$. By item (a) of Assumption 3.7 and by the definition of *clear*, p' does not contain $\texttt{emit}_{int}$ statements. Thus

$$rank(\delta) = rank(\delta') = \langle 0, n\rangle\,.$$

Case 11.1.2. $p_1 = \texttt{break}$. Similar to Case 9.1.2.

Case 11.1.3. $p_1 \neq$ `@nop`, `break`. Similar to Case 9.1.3.

Case 11.2. *isblocked*(p_1, n). Similar to Case 11.1. □

Theorem 3.14. *If* $\delta \xrightarrow[nst]{*} \delta'$ *then* $rank(\delta) \geq rank(\delta')$.

Proof. If $\delta \xrightarrow[nst]{*} \delta'$ then $\delta \xrightarrow[nst]{i} \delta'$, 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 $\delta \xrightarrow[nst]{1} \delta'_1 \xrightarrow[nst]{i-1} \delta'$, for some $i > 1$ and δ'_1. Thus, by Lemma 3.13 and by the induction hypothesis,

$$rank(\delta) \geq rank(\delta'_1) \geq rank(\delta')\,.$$ □

Theorem 3.15 (Termination)**.** *For any* δ*, there is a* $\delta'_\#$ *such that* $\delta \longrightarrow^* \delta'_\#$.

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

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

$$\langle i, j\rangle = \langle 0, 0\rangle \geq rank(\delta')\,,$$

which implies $rank(\delta') = \langle 0, 0\rangle$.

Induction. Let $\langle i, j\rangle \neq \langle 0, 0\rangle$. There are two subcases.

Case 1. δ is nested-irreducible. There are two subcases.

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

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

Case 1.2.1. $e \neq \varepsilon$. Then, by **push** and by Theorem 3.8, 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 item (a) of Lemma 3.12 and by Theorem 3.14,

$$\begin{aligned} rank(\delta) = rank(\delta'_1) &= \langle i, j\rangle \\ &\geq rank(\delta') = \langle i', j'\rangle\,. \end{aligned}$$

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

$$rank(\delta) > rank(\delta')\,.$$

By the induction hypothesis, $\delta' \longrightarrow^* \delta''_\#$, for some $\delta''_\#$. Therefore, $\delta \longrightarrow^* \delta''_\#$.

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

$$rank(\delta) > rank(\delta')\,.$$

Hence, by the induction hypothesis, there is a $\delta''_\#$ such that $\delta' \longrightarrow^* \delta''_\#$. Therefore, $\delta \longrightarrow^* \delta''_\#$.

Case 2. δ is not nested-irreducible. Then $e = \varepsilon$ and, by Theorems 3.8 and 3.14, 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. □

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. Root access is required to install a single executable file.
- **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 \
                       liblua5.3-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);
        emit B(100);
    end
end

var int a = await A; // main program
var int b = await B;

escape a+b;  // 110 is the result of the program back to the OS
```