1 Formalization of Céu

In this section, we present a formal description of Céu using an operational semantics [**?**]. We give a detailed behavior of individual computation steps, which also helps to understand our implementation, to be presented in Section **??**. The semantics is centered on the specification of a reaction chain, i.e., the program behavior while reacting to a single external event.

The semantics is split in two set of rules, which are interleaved during a reaction chain. First, we apply a single *big-step* to awake awaiting trails matching the occurring external event. Then, we continuously apply *small-steps* to handle unblocked trails until the program either terminates or awaits again in all concurrent trails. This phase may trigger internal events, which are all recorded to be applied in the next big-step. This cycle repeats until the program becomes blocked and no remaining emits are stacked for execution.

1.1 Syntax

We have carefully chosen the smallest subset of Céu that is sufficient enough to describe all peculiarities of the language:

```
// primary primitives
mem                     (any memory access)
await e                 (event await -- int/ext)
emit e                  (event emit  -- int)
break                   (loop escape)

// compound primitives
if mem then p [else q]  (conditional)
p ; q                   (sequence)
loop p                  (loop)
p and q                 (par/and)
p or q                  (par/or)

// derived by semantic rules
nop[(v)]                (terminated computation)
delay(e,i)              (delayed emit)
p @ loop p              (unwinded loop)
```

Primary primitives are the basic building blocks of the language, which are indivisible and always execute atomically. *Compound* primitives can be arbitrary compositions of primary and compound primitives.

The challenging parts of the Céu semantics reside on its control structures, hence, we are not concerned with side effects here. This way, the *mem* primitive represents assignments, expressions, and function calls, which behave as in typical imperative languages.

An *await* can refer to external or internal events, while an *emit* can only refer to internal events.

The last three helping primitives are generated by semantics rules (i.e. the programmer cannot write them): A *nop* represents a terminating computation and can be associated with a constant value. A *delay* prevents an *emit* to act immediately, deferring its execution. It is used as an artifice to give the desired *stacked* behavior for internal events. A *loop* is expanded with the special `'@'` separator (instead of `';'`) to properly bind p to it.

1.2 Small-step rules

As briefly introduced, small-step rules continuously applies transformations to unblocked trails. A program is blocked if all of its parallel branches are hanged in an *await* or *delay* primitive. The recursive predicate *isBlocked* is formally defined as follows:

$$\begin{aligned} isBlocked(p) = \ & (p = await\ e) \\ \vee\ & (p = delay\ (e, i)) \\ \vee\ & (p = q\ ;\ r)\ \wedge\ isBlocked(q) \\ \vee\ & (p = q\ @\ loop\ r)\ \wedge\ isBlocked(q) \\ \vee\ & (p = q\ and\ r)\ \wedge\ isBlocked(q)\ \wedge\ isBlocked(r) \\ \vee\ & (p = q\ or\ r)\ \wedge\ isBlocked(q)\ \wedge\ isBlocked(r) \end{aligned}$$

All small-step semantic rules are associated with a priority i, which are related to the stacked execution of internal events (to be explained in Section 1.3).

We start with the small-step rules for primary primitives:

$$mem \xrightarrow{i} nop(v) \qquad \textbf{(mem)}$$

$$emit\ e \xrightarrow{i} delay\ (e, i+1) \qquad \textbf{(emit)}$$

A *mem* terminates with a value and an *emit* is delayed with a new priority. All delayed emits are matched against awaits at once in a big-step rule, to be presented in Section 1.3.

All other primary primitives (*nop*, *break*, *await*, and *delay*) represent terminated or blocked trails, and hence, do not appear alone in the left side of a small-step rule.

The rules for conditionals are straightforward:

$$\frac{mem \xrightarrow{i} nop(v)\ ,\quad (v \neq 0)}{(if\ mem\ then\ p\ else\ q) \xrightarrow{i} p} \qquad \textbf{(if-true)}$$

$$\frac{mem \xrightarrow{i} nop(0)}{(if\ mem\ then\ p\ else\ q) \xrightarrow{i} q} \qquad \textbf{(if-false)}$$

$$(if\ mem\ then\ p) \xrightarrow{i} (if\ mem\ then\ p\ else\ nop) \qquad \textbf{(if-else)}$$

A conditional proceeds to either p or q, depending on the evaluation of *mem* (rules **if-true** and **if-false**). Note that an empty *else* branch is substituted by a *nop* (rule **if-else**).

We follow with rules for sequences:

$$\frac{p \xrightarrow{i} p'}{(p\ ;\ q) \xrightarrow{i} (p'\ ;\ q)} \quad \textbf{(seq-1)}$$

$$(nop\ ;\ q) \xrightarrow{i} q \quad \textbf{(seq-2)}$$

$$(break\ ;\ q) \xrightarrow{i} break \quad \textbf{(seq-3)}$$

Rule **seq-1** advances composition p until it becomes either a *nop* (proceeding to q in rule **seq-2**), or a *break* (ignoring q in rule **seq-3**).

Note that if p is blocked, rule **seq-1** cannot advance, as there are no rules to transform the blocked primitives *await* and *delay*.

Rules for loops are analogous to sequences, but use `'@'` as separators (to be clarified further):

$$(loop\ p) \xrightarrow{i} (p\ @\ loop\ p) \quad \textbf{(loop-exp)}$$

$$\frac{p \xrightarrow{i} p'}{(p\ @\ loop\ q) \xrightarrow{i} (p'\ @\ loop\ q)} \quad \textbf{(loop-1)}$$

$$(nop\ @\ loop\ p) \xrightarrow{i} loop\ p \quad \textbf{(loop-2)}$$

$$(break\ @\ loop\ p) \xrightarrow{i} nop \quad \textbf{(loop-3)}$$

When a program first encounters a *loop*, it first expands its body in sequence with itself (rule **loop-exp**).

Rules **loop-1** and **loop-2** are similar to rules **seq-1** and **seq-2**, advancing the loop until it reaches a *nop*. However, what follows a loop is the loop itself (rule **loop-2**).

Rule **loop-3** escapes the loop a *break* is bounded to, transforming everything into a *nop*. Note that if we used a `';'` as separators for loops, rules **loop-3** and **seq-3** would conflict.

The parallel composition *and* advances its sides independently, via rules **and-1** and **and-2**:

$$\frac{p \xrightarrow{i} p'}{(p\ and\ q) \xrightarrow{i} (p'\ and\ q)} \quad \textbf{(and-1)}$$

$$\frac{q \xrightarrow{i} q'}{(p\ and\ q) \xrightarrow{i} (p\ and\ q')} \quad \textbf{(and-2)}$$

$$(nop\ and\ q) \xrightarrow{i} q \quad \textbf{(and-3)}$$

$$(p\ and\ nop) \xrightarrow{i} p \quad \textbf{(and-4)}$$

$$\frac{q{=}break \vee isBlocked(q)}{(break\ and\ q) \xrightarrow{i} break} \quad \textbf{(and-5)}$$

$$\frac{p{=}break \vee isBlocked(p)}{(p\ and\ break) \xrightarrow{i} break} \quad \textbf{(and-6)}$$

When one of the sides of an *and* reaches a *nop*, the whole composition is substituted by the other side (rules **and-3** and **and-4**), as both trails of a `par/and` must terminate.

When one of the sides reaches a *break*, if the other side is blocked, it is killed and the whole *and* becomes a *break* (rules **and-5** and **and-6**).

For instance, the program `loop(break and (emit a))` never emits a, because the *emit* is delayed (rule **emit**) and then killed (rule **and-5**).

The rules for a parallel *or* are slightly different:

$$\frac{p \xrightarrow{i} p'}{(p \ or \ q) \xrightarrow{i} (p' \ or \ q)} \quad \textbf{(or-1)}$$

$$\frac{q \xrightarrow{i} q'}{(p \ or \ q) \xrightarrow{i} (p \ or \ q')} \quad \textbf{(or-2)}$$

$$\frac{q{=}nop \vee isBlocked(q)}{(nop \ or \ q) \xrightarrow{i} nop} \quad \textbf{(or-3)}$$

$$\frac{p{=}nop \vee isBlocked(p)}{(p \ or \ nop) \xrightarrow{i} nop} \quad \textbf{(or-4)}$$

$$\frac{q{=}nop \vee q{=}break \vee isBlocked(q)}{(break \ or \ q) \xrightarrow{i} break} \quad \textbf{(or-5)}$$

$$\frac{p{=}nop \vee p{=}break \vee isBlocked(p)}{(p \ or \ break) \xrightarrow{i} break} \quad \textbf{(or-6)}$$

Rules **or-3** and **or-4** terminate an *or* when one of the sides is a *nop*, as expected from `par/or` compositions. Note that these rules enforce the other side to advance until it either terminates or blocks.

Rules **or-5** and **or-6** are similar to their *and* counterparts, with the additional remark that a *break* has preference over a *nop*, i.e., when an *or* faces both side by side, the whole composition becomes a *break* instead of a *nop*.

As a final consideration, the pairs **and-1**/**and-2** and **or-1**/**or-2** are the only rules that bring nondeterminism to the small-step semantics.

(TODO: provar que independentemente, a transformacao termina no mesmo lugar)

1.3 Big-step rules

The big-step semantics is responsible for matching delayed emitted events with corresponding awaiting trails, all at once in a single step.

Big-step rules are associated with a tuple (E, i) that represents the set of occurring events E triggered with priority i.

At the beginning of each reaction chain, only a single external event can be handled, as required by the synchronous execution model. Hence, initially $(E, i) = (\{ext\}, 0)$, where ext is the external event triggered with priority 0.

For big-steps that follow sequences of small-steps, tuple (E, i) is acquired from the recursive function $next$:

$$next(p) = \begin{cases} (\{\}, 0), & p = (await\ e) \\ (\{e\}, i), & p = delay\ (e, i) \\ next(q), & p = (q\ ;\ r) \\ next(q), & p = (q\ @\ loop\ r) \\ (F \cup G, & \\ \quad max(j,k)), & p = (q\ and/or\ r) \wedge \\ & \quad (F, j) = next(q) \wedge \\ & \quad (G, k) = next(r) \end{cases}$$

The function returns the union of all delayed emits, which are all (necessarily) emitted with maximum priority. Note that *next* need not to be defined for unblocked primitives, as they never appear before a big-step.

We start with the big-step rules for an *await*, which either awakes or preserves it, depending on the set of current triggered events:

$$await\ e \xrightarrow{(E,i)} nop\ , \quad (e \in E) \qquad \textbf{(Await-awk)}$$
$$await\ e \xrightarrow{(E,i)} await\ e\ , \quad (e \notin E) \quad \textbf{(Await-rep)}$$

The rules for *delay* are fundamental to provide the stacked semantics for internal events:

$$delay\ (e, i) \xrightarrow{(E,i)} delay\ (\epsilon, i)\ , \quad (e \neq \epsilon) \quad \textbf{(Delay-emt)}$$
$$delay\ (\epsilon, i) \xrightarrow{(E,i)} nop \qquad \textbf{(Delay-cnt)}$$

A delayed emit is not immediately transformed into a *nop* because the just awaken trails must all react before the emit continuation proceeds (what may involve new emits to be stacked on top of it).

Hence, rule **Delay-emt** states that a delayed emit remains delayed with the same (low) priority, but does not trigger an event anymore (ϵ denotes an event that cannot be awaited).

This way, the next sequence of small-steps execute before the continuation of resolved emits, and further emits will be delayed with higher priorities (small-step rule **emit**).

Eventually, no new emits will take place, and function *next* will pop lower priority delays, providing the desired stacked execution for internal events.

To conclude the big-step semantics, the rules for compound primitives advance their subparts all at once:

$$\frac{p \xrightarrow{(E,i)} p'}{(p\ ;\ q) \xrightarrow{(E,i)} (p'\ ;\ q)} \quad \textbf{(Seq)}$$

$$\frac{p \xrightarrow{(E,i)} p'}{(p\ @\ loop\ q) \xrightarrow{(E,i)} (p'\ @\ loop\ q)} \quad \textbf{(Loop)}$$

$$\frac{p \xrightarrow{(E,i)} p' \quad q \xrightarrow{(E,i)} q'}{(p\ and\ q) \xrightarrow{(E,i)} (p'\ and\ q')} \quad \textbf{(And)}$$

$$\frac{p \xrightarrow{(E,i)} p' \quad q \xrightarrow{(E,i)} q'}{(p\ or\ q) \xrightarrow{(E,i)} (p'\ or\ q')} \quad \textbf{(Or)}$$

Note that there are no rules for *mem*, *break*, *emit*, *nop*, and *if* because none of these represent blocked primitives, and hence, never appear in a big-step.

1.4 A reaction chain

We are ready to formalize a complete reaction chain to an external event:

$$p \xrightarrow[big]{(E,i)=(\{ext\},0)} \quad [\quad (p' \xrightarrow[small]{i} p'')* \xrightarrow[big]{(E,i)=next(p)} \quad p''' \quad]*$$

A reaction chain always start with a single external event on set E. After the initial big-step, the semantics interleaves sequences of small-steps followed by a single big-step for the *next* emits.

There are two possible terminating conditions:

- A small-step rule transforms the whole program into a *nop*, meaning that the program terminated.
- Function *next* returns $(\epsilon, 0)$, meaning that program is blocked and there are no remaining emits.

(TODO: provar que isso sempre acontece)

Finally, there is a special case for the first reaction chain, which usually finds the program unblocked. We redefine every program to await the special starting event `'$'`:

$$p_0 = (await\ \$\ ;\ \ < PROG >) \qquad ext_0 = \$$$

This way, programs always start blocked and, only for the first reaction chain, we force $(E, i) = (\{\$\}, 0)$.