AN ALTERNATIVE OPERATIONAL SEMANTICS FOR CÉU

1. Abstract syntax

The *abstract syntax* of Céu programs is given by the following grammar:

$p \in P ::= \texttt{mem}(v)$	read variable v
$\mid v = x$	write x to variable v
$\mid \texttt{awaitExt}(E)$	await external event E
$\mid \texttt{awaitInt}(e)$	await internal event e
$\mid \texttt{emit}(e)$	emit event e
$\mid \texttt{break}$	break innermost loop
$\mid \texttt{if}\,\texttt{mem}(v)\,\texttt{then}\,p_1\,\texttt{else}\,p_2$	conditional
$\mid p_1; p_2$	sequence
$\mid \texttt{loop}\,p_1$	repetition
$\mid p_1\ \texttt{and}\ p_2$	par/and
$\mid p_1\ \texttt{or}\ p_2$	par/or
$\mid \texttt{fin}\,p_1$	finalization
$\mid \texttt{@awaitingExt}(E, n)$	awaiting E since reaction n
$\mid \texttt{@awaitingInt}(e, n)$	awaiting e since reaction n
$\mid \texttt{@emitting}(e, n)$	emitting e on stack level n
$\mid p_1\,\texttt{@loop}\,p_2$	unwinded loop
$\mid \texttt{skip}$	nop
$\mid \otimes$	end of a program,

where $n \in N$ is an integer, $v \in V$ is a memory location (variable) identifier, $e \in E_{internal}$ is an internal event identifier, $E \in E_{external}$ is an external event identifier and p, p_1, $p_2 \in P$ are programs. We use "`skip`" (do nothing) to represent a nop (no operation) command. "$\texttt{mem}(v)$" represents an access (read or write) to memory location v.

2. Operational Semantics: the Reaction step

For our purposes, the *state* of a Céu program within a reaction is represented by a stack S of event identifiers and a memory σ. S is composed of an initial external event, $S_0 \in E_{external}$, followed by internal events, $e_1 e_2 \ldots e_n \in E^*_{internal}$, where the leftmost element is the top-of-stack. The set $E_{events} = E_{external} \cup E_{internal}$ defines all possible event identifiers Céu programs may react to. The set Σ of memory states of Céu programs consists of all functions $\sigma : Mem \to v$, in which Mem is a memory address identifier and v represents any valid value in Céu (σ maps a memory location to a value).

A *configuration* is a triple $\langle p, \alpha, \sigma, n\rangle \in P \times E^*_{events} \times \Sigma \times N = \Delta$ that represents the situation of program p waiting to be evaluated in stack α, memory σ and reaction n. Given an initial configuration, each program reaction is computed by successive applications of the reaction-step relation $\to \subseteq \Delta \times \Delta$ such that $\langle p, \alpha, \sigma, n\rangle \to \langle p', \alpha', \sigma', n\rangle$ iff: an execution step of program p in stack α, memory σ and reaction n evaluates to a modified program p', a modified stack α'

and a modified memory σ' in the same reaction. Since relation $\rightarrow$ is defined in a way such that it can only relate configurations with the same n, we often write $\langle p, \alpha, \sigma\rangle \xrightarrow{n} \langle p', \alpha', \sigma'\rangle$ for $\langle p, \alpha, \sigma, n\rangle \rightarrow \langle p', \alpha', \sigma', n\rangle$.

The reaction-step relation $\rightarrow$ is defined inductively by the rules presented in Sections 2.1 to 2.6.

2.1. **Skip.** For all $n \in N$, $\alpha \in E^*_{events}$, and $\sigma \in \Sigma$:

$$(R_{\text{skip}}) \qquad \langle \texttt{skip}, \alpha, \sigma\rangle \xrightarrow{n} \langle \otimes, \alpha, \sigma\rangle$$

2.2. **Memory access.** For all $n \in N$, $v \in V$, $\alpha \in E^*_{events}$, and $\sigma \in \Sigma$:

$$(R_{\text{read}}) \qquad \langle \texttt{mem}(v), \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{skip}, \alpha, \sigma\rangle$$

$$(R_{\text{write}}) \qquad \langle v = x, \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{skip}, \alpha, \sigma'\rangle \qquad \text{with} \begin{cases} \sigma'(u) = \sigma(u), \text{for } u \neq v \\ \sigma'(v) = x \end{cases}$$

2.3. **Await and emit.** For all n, $n' \in N$, $E \in E_{external}$, $e \in E_{internal}$, $\alpha \in E^*_{events}$, and $\sigma \in \Sigma$:

$$(R_{\text{await-ext}}) \qquad \langle \texttt{awaitExt}(E), \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{@awaitingExt}(E, n'), \alpha, \sigma\rangle \qquad \text{with } n' = n + 1$$

$$(R_{\text{awake-ext}}) \qquad \langle \texttt{@awaitingExt}(E, n'), E\alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{skip}, E\alpha, \sigma\rangle \qquad \text{if } n' < n$$

$$(R_{\text{await-int}}) \qquad \langle \texttt{awaitInt}(e), \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{@awaitingInt}(e, n), \alpha, \sigma\rangle$$

$$(R_{\text{awake-int}}) \qquad \langle \texttt{@awaitingInt}(e, n), e\alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{skip}, e\alpha, \sigma\rangle$$

$$(R_{\text{emit-push}}) \qquad \langle \texttt{emit}(e), \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{@emitting}(n'), e\alpha, \sigma\rangle \qquad \text{with } n' = |\alpha|\,.$$

$$(R_{\text{emit-pop}}) \qquad \langle \texttt{@emitting}(n'), \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{skip}, \alpha, \sigma\rangle \qquad \text{if } n' = |\alpha|\,.$$

2.4. **Conditionals.** For all $n \in N$, $v \in V$, $\alpha \in E^*_{events}$, p_1, $p_2 \in P$ and $\sigma \in \Sigma$:

$$(R_{\text{if-true}}) \quad \langle \texttt{if}\, \texttt{mem}(v)\, \texttt{then}\, p_1\, \texttt{else}\, p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p_1, \alpha, \sigma\rangle \qquad \text{if } val(v, n, \sigma) \neq 0$$

$$(R_{\text{if-false}}) \quad \langle \texttt{if}\, \texttt{mem}(v)\, \texttt{then}\, p_1\, \texttt{else}\, p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p_2, \alpha, \sigma\rangle \qquad \text{if } val(v, n, \sigma) = 0\,,$$

where $val(v, n, \sigma)$ denotes the value of variable v in reaction n and in memory σ. For simplicity, and without loss of generality, we deal only with integer variables, i.e., $val(v, n, \sigma) \in N$, for all v, n and σ. Moreover, we assume that $val(\varepsilon, n, \sigma) = 0$, for all n and σ.

2.5. **Sequences and loops.** For all $n \in N$, $v \in V$, $\alpha, \alpha' \in E^*_{events}$, $p, p_1, p'_1, p_2 \in P$ and $\sigma, \sigma' \in \Sigma$:

$$(R_{\text{seq-skip}}) \quad \langle \texttt{skip}; p, \alpha, \sigma\rangle \xrightarrow{n} \langle p, \alpha, \sigma\rangle$$

$$(R_{\text{seq-brk}}) \quad \langle \texttt{break}; p, \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{break}, \alpha, \sigma\rangle$$

$$(R_{\text{seq-adv}}) \quad \frac{\langle p_1, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1, \alpha', \sigma'\rangle}{\langle p_1; p_2, \alpha\rangle \xrightarrow{n} \langle p'_1; p_2, \alpha', \sigma'\rangle}$$

$$(R_{\text{loop-expd}}) \quad \langle \texttt{loop}\, p, \alpha, \sigma\rangle \xrightarrow{n} \langle p\, \texttt{@loop}\, p, \alpha, \sigma\rangle$$

$$(R_{\text{loop-skip}}) \quad \langle \texttt{skip}\, \texttt{@loop}\, p, \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{loop}\, p, \alpha, \sigma\rangle$$

$$(R_{\text{loop-brk}}) \quad \langle \texttt{break}\, \texttt{@loop}\, p, \alpha, \sigma\rangle \xrightarrow{n} \langle \texttt{skip}, \alpha, \sigma\rangle$$

$$(R_{\text{loop-adv}}) \quad \frac{\langle p_1, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1, \alpha', \sigma'\rangle}{\langle p_1\, \texttt{@loop}\, p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1\, \texttt{@loop}\, p_2, \alpha', \sigma'\rangle}$$

2.6. **Par/and and par/or.** For all $n \in N$, $v \in V$, $\alpha, \alpha' \in E^*_{events}$, $p_1, p'_1, p_2, p'_2 \in P$ and $\sigma, \sigma' \in \Sigma$:

$$(R_{\text{and-skip1}}) \quad \langle \texttt{skip and}\ p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p_2, \alpha, \sigma\rangle$$

$$(R_{\text{and-skip2}}) \quad \langle p_1\ \texttt{and skip}, \alpha, \sigma\rangle \xrightarrow{n} \langle p_1, \alpha, \sigma\rangle$$

$$(R_{\text{and-brk1}}) \quad \langle \texttt{break and}\ p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_2; \texttt{break}, \alpha, \sigma\rangle \qquad \text{with } p'_2 = \kappa(p_2)$$

$$(R_{\text{and-brk2}}) \quad \langle p_1\ \texttt{and break}, \alpha\rangle \xrightarrow{n} \langle p'_1; \texttt{break}, \alpha, \sigma\rangle \qquad \text{with } p'_1 = \kappa(p_1)$$

$$(R_{\text{and-adv1}}) \quad \frac{\langle p_1, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1, \alpha', \sigma'\rangle}{\langle p_1\ \texttt{and}\ p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1\ \texttt{and}\ p_2, \alpha', \sigma'\rangle}$$

$$(R_{\text{and-adv2}}) \quad \frac{\#(p_1, \alpha, \sigma, n) \xrightarrow{n} \top \quad \langle p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_2, \alpha', \sigma'\rangle}{\langle p_1\ \texttt{and}\ p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p_1\ \texttt{and}\ p'_2, \alpha', \sigma'\rangle},$$

where the call $\kappa(p)$ (read "clear p", defined in Section **??**) concatenates in a sequence the body p' of all active finalization instructions $\texttt{fin}\, p'$ in p, and the predicate $\#(p, \alpha, \sigma, n)$ (read "p is blocked") is true iff all execution trails of p are hanged in instructions that cannot be consumed in the current reaction step, viz., `@awaitingExt`, `@awaitingInt`, `@emitting` or $\otimes$. See Appendices A.1 and A.2 for the precise definition of these functions.

The evaluation rules for par/or instructions (below) are slightly different. While in the case par/and both sides must terminate (evaluate to `skip` or `break`) for the composition to terminate, in the case of par/or the termination of *any* side causes the whole composition to terminate. Note, however, that in both cases, par/and and par/or, the right-hand side of the instruction can only be evaluated if the its left-hand side is blocked.

For all $n \in N$, $v \in V$, $\alpha, \alpha' \in E^*_{events}$, $p_1, p'_1, p_2, p'_2 \in P$ and $\sigma, \sigma' \in \Sigma$:

$$(R_{\text{or-skip1}}) \quad \langle \texttt{skip or } p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_2, \alpha, \sigma\rangle \qquad \text{with } p'_2 = \kappa(p_2)$$

$$(R_{\text{or-skip2}}) \quad \frac{\#(p_1, \alpha, \sigma, n) \xrightarrow{n} \top}{\langle p_1 \texttt{ or skip}, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1, \alpha, \sigma\rangle} \qquad \text{with } p'_1 = \kappa(p_1)$$

$$(R_{\text{or-brk1}}) \quad \langle \texttt{break or } p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_2; \texttt{break}, \alpha, \sigma\rangle \qquad \text{with } p'_2 = \kappa(p_2)$$

$$(R_{\text{or-brk2}}) \quad \frac{\#(p_1, \alpha, \sigma, n) \xrightarrow{n} \top}{\langle p_1 \texttt{ or break}, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1; \texttt{break}, \alpha, \sigma\rangle} \qquad \text{with } p'_1 = \kappa(p_1)$$

$$(R_{\text{or-adv1}}) \quad \frac{\langle p_1, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1, \alpha', \sigma'\rangle}{\langle p_1 \texttt{ or } p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_1 \texttt{ or } p_2, \alpha', \sigma'\rangle}$$

$$(R_{\text{or-adv2}}) \quad \frac{\#(p_1, \alpha, \sigma, n) \xrightarrow{n} \top \quad \langle p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p'_2, \alpha', \sigma'\rangle}{\langle p_1 \texttt{ or } p_2, \alpha, \sigma\rangle \xrightarrow{n} \langle p_1 \texttt{ or } p'_2, \alpha', \sigma'\rangle}$$

3. Properties of the reaction-step relation

Theorem 1 (Determinism)**.** *For all p, p_1, $p_2 \in P$, α, α_1, $\alpha_2 \in E^*$, and $n \in N$:*

$$(\langle p, \alpha\rangle \xrightarrow{n} \langle p_1, \alpha_1\rangle \quad \textit{and} \quad \langle p, \alpha\rangle \xrightarrow{n} \langle p_2, \alpha_2\rangle) \quad \textit{implies} \quad \langle p_1, \alpha_1\rangle = \langle p_2, \alpha_2\rangle\,.$$

Proof. Suppose $\langle p, \alpha\rangle \xrightarrow{n} \langle p_1, \alpha_1\rangle$ and $\langle p, \alpha\rangle \xrightarrow{n} \langle p_2, \alpha_2\rangle$, for arbitrary p, p_1, $p_2 \in P$, α, α_1, $\alpha_2 \in E^*$, and $n \in N$. We proceed by structural induction on p. The following ten cases are possible. (The cases where $p = \texttt{mem}(v)$, $p = \texttt{break}$, and $p = \texttt{fin}\,p'$ are not considered since there are no rules to evaluate them, i.e., by the theorem hypothesis, $p \neq \texttt{mem}(v)$, $p \neq \texttt{break}$, and $p \neq \texttt{fin}\,p'$.)

[Case 1] $p = \texttt{awaitExt}(e)$, for some $e \in E$. By rule R_{await}, $p_1 = p_2 = \texttt{@awaitingExt}(e, n')$, with $n' = n + 1$, and $\alpha_1 = \alpha_2 = \alpha$.

[Case 2] $p = \texttt{@awaitingExt}(e, n')$, for some $e \in E$. By rule R_{awake}, $p_1 = p_2 = \texttt{skip}$, with $n' \leq n$, and $\alpha_1 = \alpha_2 = \alpha$.

[Case 3] $p = \texttt{emit}(e)$, for some $e \in E$. ...

[Case 4] $p = \texttt{@emitting}(n')$, for some $n' \in N$. ...

[Case 5] $p = \texttt{if mem}(v)\,\texttt{then}\,p'\,\texttt{else}\,p''$, for some $v \in V$ and $p', p'' \in P$. ...

[Case 6] $p = p'; p''$, for some $p', p'' \in P$. ...

[Case 7] $p = \texttt{loop}\,p'$, for some $p' \in P$. ...

[Case 8] $p = p'\,\texttt{@loop}\,p''$, for some $p', p'' \in P$. ...

[Case 9] $p = p' \texttt{ and } p''$, for some $p', p'' \in P$. ...

[Case 10] $p = p' \texttt{ or } p''$, for some $p', p'' \in P$. ...

TODO. Aqui eu esbarro no primeiro problema. A indução em P fica complicada porque há programas que não avaliam para nada. Por exemplo, se $p = \texttt{if mem}(v)\,\texttt{then}\,p'\,\texttt{else}\,p''$ com $p' = \texttt{break}$, então a hipótese indutiva não vale p', mesmo com $p' \prec p$. Ou seja, eu teria que identificar todos os casos em que o p' da hipótese indutiva não vale. Eu vejo duas soluções simples para esse problema.

A primeira solução é fazer com que todo programa avalie para alguma coisa, e.g., definir regras do tipo:

$$\langle \texttt{mem}(v), \alpha \rangle \xrightarrow{n} \langle \texttt{skip}, \alpha \rangle$$
$$\langle \texttt{break}, \alpha \rangle \xrightarrow{n} \langle \texttt{break}, \alpha \rangle$$
$$\langle \texttt{fin}\, p_1, \alpha \rangle \xrightarrow{n} \langle \texttt{fin}\, p_1, \alpha \rangle .$$

Nesse caso, teríamos que ver se as regras acima são suficientes e se elas não causam conflitam com as existentes.

A segunda solução, é deixar em P apenas os programas que avaliam para alguma coisa. Ou seja, $\texttt{mem}(v)$ e $\texttt{break}$ não fariam parte de P mas sim de um conjunto auxiliar. Dessa forma, quando eu fizesse uma indução em P não teria que lidar com esses casos. Voltado ao exemplo, $\texttt{if}\,\texttt{mem}(v)\,\texttt{then}\,\texttt{break}\,\texttt{else}\,p''$ seria tratado como um caso base, já que na gramática eu teria algo do tipo:

$$p ::= \texttt{if}\,\texttt{mem}(v)\,\texttt{then}\,\texttt{break}\,\texttt{else}\,p_2 \mid \cdots$$

A desvantagem dessa segunda opção é que o número de regras pode aumentar significativamente.

Qualquer que seja a solução, se eu tenho $\langle p, \alpha \rangle$ sempre avaliando para alguma coisa, na hora de fazer o fecho (i.e., na definição da relação $\Rightarrow$) eu não preciso do fecho reflexivo—o transitivo basta. □

Theorem 2 (Termination)**.** *For all* $\langle p, \alpha \rangle \in \Delta$,

$$(\exists! \langle p', \alpha' \rangle \in \Delta)(\langle p, \alpha \rangle \xrightarrow{n} \langle p', \alpha' \rangle).$$

Proof. By structural induction on P.

TODO. Aqui eu caio no memso problema anterior. Há programas em P que não avaliam para nada. □

4. Reaction

TODO. Eu tinha definido a seguinte regra:

> For all $n \in N$, $e \in E$, $\alpha \in E^*$, and $p \in P$:
>
> $$(R_{\text{pop}}) \qquad \langle p, e\alpha \rangle \xrightarrow{n} \langle p, \alpha \rangle \qquad \text{if } \#(p, e\alpha, n) .$$
>
> This rule is necessary to ensure that pending `@emitting` instructions (in lower stack indexes) are eventually resumed by subsequent reaction steps.

Dessa forma, o "pop" faria parte da própria definição de passo. Mas depois eu percebi que essa regra complica (inviabiliza?) qualquer indução simples em P (acho que eu teria que fazer uma indução em pares $\langle p, \alpha \rangle$). Talvez ela tenha que ficar fora mesmo—i.e., na definição da relação $\Rightarrow$, igual o original—ou talvez haja alguma forma de implementá-la em $\rightarrow$ sem complicar as provas.

The reaction-step relation ($\rightarrow$), defined in Section 2 computes a single step within a reaction. To compute a full reaction we now define the reaction relation $\overset{n}{\Rightarrow} \subseteq \Delta \times P$ such that $\langle p, \alpha \rangle \overset{n}{\Rightarrow} p'$ iff program p in initial state α evaluates (reacts) to a modified

program p' in the empty state in a finite number of steps. More formally, for all $\alpha \in E$ and $p, p' \in P$:

$$\langle p, \alpha \rangle \overset{n}{\Rightarrow} p' \quad \text{iff} \quad \langle p, \alpha \rangle \overset{n'}{\underset{*}{\rightarrow}} \langle p, \varepsilon \rangle ,$$

where $\underset{*}{\rightarrow}$ is the reflexive-transitive closure of relation $\rightarrow$. We use $\langle p, \alpha \rangle \overset{n}{\Rightarrow} p'$ as an abbreviation for $\langle p, \alpha, n \rangle \Rightarrow p'$.

Finally, we describe the complete execution of a Céu program, given some sequence of external events, by applying the reaction relation repeatedly to each event in the sequence, with the reaction number incremented after each application.

5. Properties of the reaction relation

Theorem 3 (Determinism)**.** *For all p, p_1, $p_2 \in P$, $\alpha \in E^*$, and $n \in N$:*

$$(\langle p, \alpha \rangle \overset{n}{\Rightarrow} p_1 \quad \textit{and} \quad \langle p, \alpha \rangle \overset{n}{\Rightarrow} p_2) \quad \textit{implies} \quad p_1 = p_2 .$$

Proof. By induction on the number of reaction steps. ... □

Theorem 4 (Termination)**.** *For all p, $p' \in P$, $\alpha \in E^*$, and $n \in N$:*

$$(\exists! p' \in P)(\langle p, \alpha \rangle \overset{n}{\Rightarrow} p') .$$

Proof. By induction on the number of reaction steps. ...

TODO. Talvez eu tenha que partir para uma indução em pares ordenados $\langle p, \alpha \rangle$, já que tanto p pode diminur quanto α. □

6. Alternative formulation (Pure Céu)

TODO. Uma abordagem alternativa seria simplificar a formalização. Minha proposta é, num primeiro momento, definir uma versão reduzida (pura) de Céu retirando da linguagem as construções `mem`(v) e `if mem`(v) `then` p_1 `else` p_2 e talvez outras coisas, e.g., loop e finalização. E dado essa versão pura, tentar definir uma semântica big-step que se comporte da maneira desejada, i.e., avaliação da esquerda para direita com instruções *await* atrasadas. (Eu acho mais fácil trabalhar com a big-step, mas se não tiver jeito a gente volta para a small-step.)

Num segundo momento, minha idéia é pegar as semânticas construtivas de Esterel e ver como eles resolvem o problema do loop causal no *await* que acorda no mesmo ciclo. Talvez seja interessante não atrasar os *awaits* em Céu.

Feito isso, i.e., definida uma base semântica e provados o determinismo e convergência, eu posso partir para as provas de equivalência entre programas e, quem sabe, encontrar possíveis formas normais.

(A propósito, do jeito que está definido hoje as reações de Céu não retornam nada para o ambiente, i.e., não há evento externo de saída. É isso mesmo?)

Appendix A. Auxiliary definitions

A.1. **The clear function.** The clear function $\kappa\colon P \to P'$ concatenates in a sequence the body of all active finalization instructions $\texttt{fin}\,p'$ within some program p. Function κ is defined by recursion on P as follows:

$$
\begin{aligned}
\kappa(\texttt{skip}) &= \texttt{skip}\\
\kappa(\texttt{mem}(v)) &= \texttt{skip}\\
\kappa(\texttt{awaitExt}(e)) &= \texttt{skip}\\
\kappa(\texttt{awaitInt}(e)) &= \texttt{skip}\\
\kappa(\texttt{emit}(e)) &= \texttt{skip}\\
\kappa(\texttt{break}) &= \texttt{skip}\\
\kappa(\texttt{if}\,\texttt{mem}(v)\,\texttt{then}\,p_1\,\texttt{else}\,p_2) &= \texttt{skip}\\
\kappa(p_1; p_2) &= \kappa(p_1)\\
\kappa(\texttt{loop}\,p_1) &= \texttt{skip}\\
\kappa(p_1\ \texttt{and}\ p_2) &= \kappa(p_1); \kappa(p_2)\\
\kappa(p_2\ \texttt{or}\ p_2) &= \kappa(p_1); \kappa(p_2)\\
\kappa(\texttt{fin}\,p_1) &= p_1\\
\kappa(\texttt{@awaitingExt}(e, n)) &= \texttt{skip}\\
\kappa(\texttt{@awaitingInt}(e, n)) &= \texttt{skip}\\
\kappa(\texttt{@emitting}(n)) &= \texttt{skip}\\
\kappa(p_1\,\texttt{@loop}\,p_2) &= \texttt{skip}
\end{aligned}
$$

The form of program $p' \in P' \subseteq P$ returned by κ is given by the grammar:

$$p' \in P' ::= \texttt{skip} \mid p'_1; p'_2\,.$$

(Note that the definition of κ assumes that the body p_1 of all instructions $\texttt{fin}\,p_1$ is necessarily a member of P', i.e., p_1 is either $\texttt{skip}$ or $\texttt{mem}(v)$ or some finite sequence of such instructions.)

A.2. **The is-blocked predicate.** The is-blocked predicate $\#\colon P \times E^*_{events} \times \Sigma \times N \to \{\top, \bot\}$ checks if all execution trails of a given program p in stack α, memory σ and reaction n are blocked, i.e., if all its trails are hanged in instructions `@awaitingExt` `@awaitingInt` `@emitting` or $\otimes$, which cannot be consumed in the current reaction step. Predicate # is defined by recursion on P as follows:

$$(R_\#) \qquad \langle \texttt{skip}, \alpha, \sigma, n \rangle \to \bot$$

$$(R_\#) \qquad \langle \otimes, \alpha, \sigma \rangle \xrightarrow{n} \top$$

$$(R_\#) \qquad \frac{e' \neq e}{\langle \texttt{@awaitingExt}(e', n'), e\alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{n' = n}{\langle \texttt{@awaitingExt}(e', n'), e\alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{e' \neq e}{\langle \texttt{@awaitingInt}(e', n'), e\alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{n' \neq |\alpha|}{\langle \texttt{@emitting}(n'), \alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{\#(p_1, \alpha, \sigma, n) \xrightarrow{n} \top}{\langle p_1; p_2, \alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{\#(p_1, \alpha, \sigma, n) \xrightarrow{n} \top}{\langle p_1 \ \texttt{@loop} \ p_2, \alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{\#(p_1, \alpha, \sigma, n) \xrightarrow{n} \top \qquad \#(p_2, \alpha, \sigma, n) \xrightarrow{n} \top}{\langle p_1 \ \texttt{and} \ p_2, \alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{\#(p_1, \alpha, \sigma) \xrightarrow{n} \top \qquad \#(p_2, \alpha, \sigma) \xrightarrow{n} \top}{\langle p_1 \ \texttt{or} \ p_2, \alpha, \sigma \rangle \xrightarrow{n} \top}$$

$$(R_\#) \qquad \frac{p = \texttt{mem}(v)}{\langle p, \alpha, \sigma \rangle \xrightarrow{n} \bot}$$

$$(R_\#) \qquad \frac{p = \texttt{awaitExt}}{\langle p, \alpha, \sigma \rangle \xrightarrow{n} \bot}$$

$$(R_\#) \qquad \frac{p = \texttt{awaitInt}}{\langle p, \alpha, \sigma \rangle \xrightarrow{n} \bot}$$

$$(R_\#) \qquad \frac{p = \texttt{emit}}{\langle p, \alpha, \sigma \rangle \xrightarrow{n} \bot}$$

$$(R_\#) \qquad \frac{p = \texttt{break}}{\langle p, \alpha, \sigma \rangle \xrightarrow{n} \bot}$$

$$(R_\#) \qquad \frac{p = \texttt{if}\,\texttt{mem}(v)\,\texttt{then}\,p_1\,\texttt{else}\,p_2}{\langle p, \alpha, \sigma \rangle \xrightarrow{n} \bot}$$

$$(R_\#) \qquad \frac{p = \texttt{loop}\ p_1}{\langle p, \alpha, \sigma \rangle \xrightarrow{n} \bot}$$