VI
The semantics of Céu

The disciplined synchronous execution of Céu, together with broadcast communication and stacked execution for internal events, may raise doubts about the precise execution of programs. In this chapter, we introduce a reduced syntax of Céu and propose an operational semantics in order to formally describe the language, eliminating imprecisions with regard to how a program reacts to an external event. For the sake of simplicity, we focus on the control aspects of the language, leaving out side effects and *C* calls (which behave like in any conventional imperative language).

VI.1 Abstract syntax

```
                                  // primary expressions
p ::= mem(id)                     (any memory access to ‘id’)
    | await(id)                   (await event ‘id’)
    | emit(id)                    (emit event ‘id’)
    | break                       (loop escape)
                                  // compound expressions
    | if mem(id) then p else p    (conditional)
    | p ; p                       (sequence)
    | loop p                      (repetition)
    | p and p                     (par/and)
    | p or p                      (par/or)
    | fin p                       (finalization)
                                  // derived by semantic rules
    | awaiting(id,n)              (awaiting ‘id’ since sequence number ‘n’)
    | emitting(n)                 (emitting on stack level ‘n’)
    | p @ loop p                  (unwinded loop)
```

Figure VI.1: Reduced syntax of Céu.

Figure VI.1 shows the BNF-like syntax for a subset of Céu that is sufficient to describe all semantic peculiarities of the language. The $mem(id)$ primitive represents all accesses, assignments, and *C* function calls that affect a memory location identified by *id*. As the challenging parts of Céu reside on its control structures, we are not concerned here with a precise semantics

for side effects, but only with their occurrences in programs. The special notation *nop* is used to represent an innocuous *mem* expression (it can be thought as a synonym for $mem(\epsilon)$, where ϵ is an unused identifier). Except for the fin and semantic-derived expressions, which are discussed further, the other expressions map to their counterparts in the concrete language in Figure III.1. Note that *mem* expressions cannot share identifiers with *await*/*emit* expressions.

VI.2 Operational semantics

The core of our semantics is a relation that, given a sequence number n identifying the current reaction chain, maps a program p and a stack of events S in a single step to a modified program and stack:

$$\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle$$

where

$$\begin{aligned} S, S' \in id^* \quad & (sequence\ of\ event\ identifiers : [id_{top}, ..., id_1]) \\ p, p' \in P \quad & (as\ described\ in\ Figure\ VI.1) \\ n \in \mathbb{N} \quad & (univocally\ identifies\ a\ reaction\ chain) \end{aligned}$$

At the beginning of a reaction chain, the stack is initialized with the occurring external event *ext* ($S = [ext]$), but *emit* expressions can push new events on top of it (we discuss how they are popped further).

We describe this relation with a set of *small-step* structural semantics rules, which are built in such a way that at most one transition is possible at any time, resulting in deterministic reaction chains. The transition rules for the primary expressions are as follows:

$$\langle S,\ await(id)\rangle \xrightarrow[n]{} \langle S,\ awaiting(id, n)\rangle \qquad \textbf{(await)}$$

$$\langle id : S,\ awaiting(id, m)\rangle \xrightarrow[n]{} \langle id : S,\ nop\rangle,\ \ m < n \qquad \textbf{(awaiting)}$$

$$\langle S,\ emit(id)\rangle \xrightarrow[n]{} \langle id : S,\ emitting(|S|)\rangle \qquad \textbf{(emit)}$$

$$\langle S,\ emitting(|S|)\rangle \xrightarrow[n]{} \langle S,\ nop\rangle \qquad \textbf{(emitting)}$$

An *await* is simply transformed into an *awaiting* that remembers the current external sequence number n (rule **await**). An *awaiting* can only transit to a *nop* (rule **awaiting**) if its referred event *id* matches the top of the stack and its sequence number is smaller than the current one ($m < n$). An *emit* transits to an *emitting* holding the current stack level ($|S|$ stands for the stack length), and pushing the referred event on the stack (in rule **emit**). With the new stack level $|S|+1$, the $emitting(|S|)$ itself cannot transit, as rule **emitting** expects its parameter to match the current stack level. This trick provides the desired stack-based semantics for internal events.

Proceeding to compound expressions, the rules for conditionals and sequences are straightforward:

$$\frac{val(id,n) \neq 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \underset{n}{\longrightarrow} \langle S, p\rangle} \quad \textbf{(if-true)}$$

$$\frac{val(id,n) = 0}{\langle S, (if\ mem(id)\ then\ p\ else\ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle} \quad \textbf{(if-false)}$$

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p'\ ;\ q)\rangle} \quad \textbf{(seq-adv)}$$

$$\langle S, (mem(id)\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle \quad \textbf{(seq-nop)}$$

$$\langle S, (break\ ;\ q)\rangle \underset{n}{\longrightarrow} \langle S, break\rangle \quad \textbf{(seq-brk)}$$

Given that our semantics focuses on control, rules **if-true** and **if-false** are the only to query *mem* expressions. The "magical" function *val* receives the memory identifier and current reaction sequence number, returning the current memory value. Although the value is arbitrary, it is unique in a reaction chain, because a given expression can execute only once within it (remember that *loops* must contain *awaits* which, from rule **await**, cannot awake in the same reaction they are reached).

The rules for loops are analogous to sequences, but use `'@'` as separators to properly bind breaks to their enclosing loops:

$$\langle S, (loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, (p\ @\ loop\ p)\rangle \qquad \textbf{(loop-expd)}$$

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p\ @\ loop\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p'\ @\ loop\ q)\rangle} \qquad \textbf{(loop-adv)}$$

$$\langle S, (mem(id)\ @\ loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, loop\ p\rangle \qquad \textbf{(loop-nop)}$$

$$\langle S, (break\ @\ loop\ p)\rangle \underset{n}{\longrightarrow} \langle S, nop\rangle \qquad \textbf{(loop-brk)}$$

When a program first 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 they reach a $mem(id)$. 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 parallel *and* compositions force transitions on the left branch p to occur before transitions on the right branch q (rules **and-adv1** and **and-adv2**). Then, if one of the sides terminates, the composition is simply substituted by the other side (rules **and-nop1** and **and-nop2**):

$$
\begin{aligned}
isBlocked(n, a : S, awaiting(b, m)) &= (a \neq b \ \vee \ m = n) \\
isBlocked(n, S, emitting(s)) &= (|S| \neq s) \\
isBlocked(n, S, (p \ ; \ q)) &= isBlocked(n, S, p) \\
isBlocked(n, S, (p \ @ \ loop \ q)) &= isBlocked(n, S, p) \\
isBlocked(n, S, (p \ and \ q)) &= isBlocked(n, S, p) \wedge isBlocked(n, S, q) \\
isBlocked(n, S, (p \ or \ q)) &= isBlocked(n, S, p) \wedge isBlocked(n, S, q) \\
isBlocked(n, S, \_) &= false \ \ (mem, await, \\
&\qquad emit, break, if, loop)
\end{aligned}
$$

Figure VI.2: The recursive predicate *isBlocked* is true only if all branches in parallel are hanged in *awaiting* or *emitting* expressions that cannot transit.

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p \ and \ q)\rangle \underset{n}{\longrightarrow} \langle S', (p' \ and \ q)\rangle} \qquad \textbf{(and-adv1)}$$

$$\frac{isBlocked(n, S, p) \ , \ \langle S, q\rangle \underset{n}{\longrightarrow} \langle S', q'\rangle}{\langle S, (p \ and \ q)\rangle \underset{n}{\longrightarrow} \langle S', (p \ and \ q')\rangle} \qquad \textbf{(and-adv2)}$$

$$\langle S, (mem(id) \ and \ q)\rangle \underset{n}{\longrightarrow} \langle S, q\rangle \qquad \textbf{(and-nop1)}$$

$$\langle S, (p \ and \ mem(id))\rangle \underset{n}{\longrightarrow} \langle S, p\rangle \qquad \textbf{(and-nop2)}$$

$$\langle S, (break \ and \ q)\rangle \underset{n}{\longrightarrow} \langle S, (clear(q) \ ; \ break)\rangle \qquad \textbf{(and-brk1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p \ and \ break)\rangle \underset{n}{\longrightarrow} \langle S, (clear(p) \ ; \ break)\rangle} \qquad \textbf{(and-brk2)}$$

The deterministic behavior of the semantics relies on the *isBlocked* predicate, defined in Figure VI.2 and used in rule **and-adv2**, requiring the left branch p to be blocked in order to allow the right transition from q to q'. An expression becomes blocked when all of its trails in parallel hang in *awaiting* and *emitting* expressions.

The last two rules **and-brk1** and **and-brk2** deal with a *break* in each of the sides in parallel. A *break* should terminate the whole composition in order

$$
\begin{aligned}
clear(fin\ p) &= p \\
clear(p\ ;\ q) &= clear(p) \\
clear(p\ @\ loop\ q)) &= clear(p) \\
clear(p\ and\ q) &= clear(p)\ ;\ clear(q) \\
clear(p\ or\ q) &= clear(p)\ ;\ clear(q) \\
clear(\_) &= mem(id)
\end{aligned}
$$

Figure VI.3: The function *clear* extracts fin expressions in parallel and put their bodies in sequence.

to escape the innermost loop (*aborting* the other side). The *clear* function in the rules, defined in Figure VI.3, concatenates all active fin bodies of the side being aborted (to execute before the *and* rejoins). Note that there are no transition rules for fin expressions. This is because once reached, an fin expression only executes when it is aborted by a trail in parallel. In Section VI.3(c), we show how an fin is mapped to a finalization block in the concrete language. Note that there is a syntactic restriction that an fin body cannot *emit* or *await*—they are guaranteed to completely execute within a reaction chain.

Most rules for parallel *or* compositions are similar to *and* compositions:

$$\frac{\langle S, p\rangle \underset{n}{\longrightarrow} \langle S', p'\rangle}{\langle S, (p\ or\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p'\ or\ q)\rangle} \qquad \textbf{(or-adv1)}$$

$$\frac{isBlocked(n, S, p)\ ,\ \ \langle S, q\rangle \underset{n}{\longrightarrow} \langle S', q'\rangle}{\langle S(p\ or\ q)\rangle \underset{n}{\longrightarrow} \langle S', (p\ or\ q')\rangle} \qquad \textbf{(or-adv2)}$$

$$\langle S, (mem(id)\ or\ q)\rangle \underset{n}{\longrightarrow} \langle S, clear(q)\rangle \qquad \textbf{(or-nop1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p\ or\ mem(id))\rangle \underset{n}{\longrightarrow} \langle S, clear(p)\rangle} \qquad \textbf{(or-nop2)}$$

$$\langle S, (break\ or\ q) \underset{n}{\longrightarrow} \langle S, (clear(q)\ ;\ break)\rangle \qquad \textbf{(or-brk1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p\ or\ break)\rangle \underset{n}{\longrightarrow} \langle S, (clear(p)\ ;\ break)\rangle} \qquad \textbf{(or-brk2)}$$

For a parallel *or*, the rules **or-nop1** and **or-nop2** must terminate the composition, and also apply the function *clear* to the aborted side, in order to properly finalize it.

A reaction chain eventually blocks in *awaiting* and *emitting* expressions in parallel trails. If all trails hangs only in *awaiting* expressions, it means that the program cannot advance in the current reaction chain. However, *emitting* expressions should resume their continuations of previous *emit* in the ongoing reaction, they are just hanged in lower stack indexes (see rule **emit**). Therefore, we define another relation that behaves as the previous if the program is not blocked, and, otherwise, pops the stack:

$$\frac{\langle S,p\rangle \underset{n}{\longrightarrow} \langle S',p'\rangle}{\langle S,p\rangle \underset{n}{\Longrightarrow} \langle S',p'\rangle} \qquad \frac{isBlocked(n,\ s:S,\ p)}{\langle s:S,p\rangle \underset{n}{\Longrightarrow} \langle S,p\rangle}$$

To describe a *reaction chain* in Céu, i.e., how a program behaves in reaction to a single external event, we use the reflexive transitive closure of this relation:

$$\langle S,p\rangle \overset{*}{\underset{n}{\Longrightarrow}} \langle S',p'\rangle$$

Finally, to describe the complete execution of a program, we need multiple "invocations" of reaction chains, incrementing the sequence number:

$$\langle [e1],p\rangle \overset{*}{\underset{1}{\Longrightarrow}} \langle [],p'\rangle$$
$$\langle [e2],p'\rangle \overset{*}{\underset{2}{\Longrightarrow}} \langle [],p''\rangle$$
$$\ldots$$

Each invocation starts with an external event at the top of the stack and finishes with a modified program and an empty stack. After each invocation, the sequence number is incremented.

VI.3 Concrete language mapping

Although the reduced syntax presented in Figure VI.1 is similar to the concrete language in Figure III.1, there are some significant mismatches between Céu and the formal semantics that require some clarification. In this section, we describe an informal mapping between the two.

Most statements from Céu map directly to the formal semantics, e.g., `if` $\mapsto$ *if*, `';'` $\mapsto$ *';'*, `loop` $\mapsto$ *loop*, `par/and` $\mapsto$ *and*, `par/or` $\mapsto$ *or*. (Again, we are not considering side-effects, which are all mapped to the *mem* semantic construct.)

(a) await and emit

The `await` and `emit` primitives of CÉU are slightly more complex in comparison to the formal semantics, as they support communication of values between emits and awaits. In the two-step translation below, we start with the program in CÉU, which communicates the value 1 between the `emit` and `await` in parallel (left-most code). In the intermediate translation, we include the shared variable `e_` to hold the value being communicated between the two trails in order to simplify the `emit`. Finally, we convert the program into the equivalent in the formal semantics, translating side-effect statements into *mem* expressions:

```
par/or do
  <...>
  emit e => 1;
with
  v = await e;
  _printf("%d\n",v);
end
```

```
par/or do
  <...>
  e_ = 1;
  emit e;
with
  await e;
  v = e_;
  _printf("%d\n",v);
end
```

```
<...> ; mem ; emit(e)
        or
await(e) ; mem ; mem
```

Note that a similar translation is required for external events, i.e., each external event has a corresponding variable that is explicitly set by the environment before each reaction chain.

(b) First-class timers

To encompass first-class timers, we need a special `TICK` event that should be intercalated with each other event occurrence in an application (e.g. *e1*, *e2*):

$$\langle [TICK], p \rangle \xRightarrow[1]{*} \langle [], p' \rangle$$
$$\langle [e1], p' \rangle \xRightarrow[2]{*} \langle [], p'' \rangle$$
$$\langle [TICK], p'' \rangle \xRightarrow[3]{*} \langle [], p''' \rangle$$
$$\langle [e2], p''' \rangle \xRightarrow[4]{*} \langle [], p'''' \rangle$$
$$\ldots$$

The `TICK` event has an associated variable `TICK_` (as illustrated in the previous section) with the time elapsed between the two occurrences of external events.

The translation in two steps from a timer await to the semantics is as follows:

```
dt = await 10ms;
```

```
var int tot = 10000;
loop do
    await TICK;
    tot = tot - TICK_;
    if tot <= 0 then
        dt = tot;
        break;
    end
end
```

```
mem;
loop(
    await(TICK);
    mem;
    if mem then
        mem;
        break
    else
        nop
)
```

(c) Finalization blocks

The biggest mismatch between CÉU and the formal semantics is regarding finalization blocks, which require more complex modifications in the program for a proper mapping using the *fin* semantic construct. The code that follows uses a `finalize` to safely `_release` the reference to `ptr` kept after the call to `_hold`:

```
do
    var int* ptr = <...>;
    await A;
    finalize
        _hold(ptr);
    with
        _release(ptr);
    end
    await B;
end
```

In the translation to the semantics, the first required modification is to catch the `do-end` termination to run the finalization code. For this, we translate the block into a `par/or` with the original body in parallel with a *fin* to run the finalization code:

```
par/or do
    var int* ptr = <...>;
    await A;
    _hold(ptr);
    await B;
with
    { fin
        _release(ptr); }
end
```

In this intermediate code (mixing the syntaxes), the *fin* body will execute whenever the `par/or` terminates, either normally (after the `await B`) or aborted from an outer composition (rules **and-brk1**, **and-brk2**, **or-nop1**, **or-nop2**, **or-brk1**, and **or-brk2** in the semantics). However, the *fin* will also (incorrectly) execute even if the call to `_hold` is not reached in the body due to an abort before awaking from the `await A`. To deal with this issue, for each *fin* we need a corresponding flag to keep track of code that needs to be finalized:

```
1  f_ = 0;
2  par/or do
3      var int* ptr = <...>;
4      await A;
5      _hold(ptr);
6      f_ = 1;
7      await B;
8  with
9      { fin
10         if f_ then
11             _release(ptr);
12         end }
13 end
```

The flag is initially set to false (line 1), avoiding the finalization code to execute (lines 9-12). Only after the call to `_hold` (line 5) that we set the flag to true (line 6) and enable the *fin* body to execute. The complete translation from the original example in CÉU is as follows:

```
mem;     // f_ = 0
(
   mem;          // ptr = <...>
   await(A);
   mem;          // _hold(ptr)
   mem;          // f_ = 1
   await(B);
or
   fin
     if mem then    // if f_
        mem         // release _ptr
     else
        nop
)
```