Valued Signals vs. Variables

- **Data expressions:**
  - references to constants or variables
  - $?S$ yields the current value of signal $S$
  - $\text{pre}(?S)$ yields the value of signal $S$ at the previous instant

- **Assignment (instantaneous):**
  $X := e$ where $X$ is a variable and $e$ is a data expression

- **emit $S(e)$:** evaluates the data expression $e$, emits $S$ with that value and terminates instantaneously.
## Valued Signals vs. Variables

<table>
<thead>
<tr>
<th>The value of a signal can be changed only if the status is <em>present</em>.</th>
<th>The value of a variable is written by an instantaneous assignment statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlike the status, the value is permanent: if it is unchanged in an instant, its value is that of the previous instant.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A valued signal has exactly one status and exactly one value at a time (per instant).</th>
<th>A variable can take several successive values in an instant.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both the status and the value are broadcast.</td>
<td>Order in which the values are taken: constructive order.</td>
</tr>
</tbody>
</table>

| A signal is shared throughout its scope. | A variable is local to a thread in case the thread writes it. If the thread forks on `||`, only two cases are legal: |
|---|---|
| | The variable is accessed in read-only mode in each subthread, or |
| | if the variable is written by some thread, then it can neither be read nor be written by concurrent threads. |
Valued Signals vs. Variables

- \( X := X + 1 \oplus X := 1 \)  
  - forbidden

- \( \text{emit (S1)} \oplus \text{emit (S2)} \)  
  - allowed

- \( X := X + 1 \)  
  - allowed

- \( \text{emit S(}$?S+1$\text{)} \)  
  - forbidden
Example (1)

module System1:
input A, B, R;
output O;
loop
    [await A || await B]
    emit O
each R
end module
Example (2)

- every S do p end awaits the first future occurrence (ie not at initialization time) of S to start p.

- every immediate S do p end immediately starts p if I is present at the first instant.

```plaintext
module Count1:
    input I;
    output COUNT:=0:integer;
    every I do
        emit COUNT(pre(?COUNT)+1);
    end every
end module

module Count2:
    input I;
    output COUNT;
    var Count:=0:integer in
        every I do
            Count:=Count+1;
            emit(COUNT(Count))
        end every
end var
end module
```
Abort

- Behavior of abort p when S:
  - In the starting instant, p is immediately started, the initial presence or absence of S being ignored (delayed abort).
  - If p terminates before S occurs, then the whole abort statement terminates.
  - If S occurs while p is not yet terminated, the abort statement immediately terminates and p does not receive control in the current instant (strong abort).
- To make abort sensitive to S in the first instant:
  abort p when immediate S
- To give p control a last time when S occurs:
  weak abort p when S

module Speedometer:
input Second, Meter;
output Speed: integer in
loop
  var distance:=0: integer in
  abort
    every Meter do
      distance:=distance+1
    end every
  when Second do
    emit Speed(distance)
  end abort
end var
end loop
end module
Generic Behaviors and Modules

- Each data object used by a module must be declared in that module.
- A data object defined in different submodules must be identically declared.
- Calling modules: run statement. Explicit renaming by '/'.
- Renaming arguments are passed by name and not by position!
- If a name is kept unchanged in a substitution, it need not be passed as a parameter.

```plaintext
module TWO_STATES:
  input On, Off;
  output IsOn, IsOff;
  loop
    abort
    sustain IsOff
    when On
    abort
    sustain IsOn
    when Off
  end loop
end module

run TWO STATES [signal RadioOn / On, RadioOff / Off, Playing / IsOn]
```

end
ESTEREL – Example Program

module Speedometer:
  input Second, Meter;
  output Speed: integer in
  loop
    var distance:=0: integer in
    weak abort
      every Meter do
        distance:=distance+1
      end every
    when Second do
      emit Speed(distance)
    end abort
  end var
end loop
end module

module SpeedSupervisor:
  input Second, Meter;
  output TooFast in
  signal Speed: integer in
  \ run Speedometer
  ||
  \ every Speed do
  \   if ?Speed > MaxSpeed
  \     then emit TooFast
  \   end if
  \ end every
  ]
end signal
end module
Causality Analysis

- **Causality problem**: the presence of a signal seems to depend on itself (problem of combinatorial loops in synchronous circuits).
- **Goal**: have one (reactivity) and only one (determinism) consistent solution for each configuration of input signals.
- **Example situations**:

```plaintext
module P3:
  input I;
  output O;
  signal S in
    present I then emit S end
  ||
    present S then emit O end
  end signal
end module

module P1:
  output O;
  present O
    else emit O end present
end module

module P2:
  output O;
  present O
    then emit O end present
end module

inconsistent non-deterministic logically correct
```
Causality Analysis

module P4:
output O1,O2;
present O1 then emit O1 end
||
present O1 then
  present O2 else emit O2 end
end present
end module

Logically correct, but rejected by Constructive Causality:
no constructive explanation for solution.
Logical Correctness

- **Logical coherence law**: A signal $S$ is present in an instant if and only if an `emit S` statement is executed in this instant.

- Logical correctness requires: there exists exactly one status for each signal that respects the coherence law.

- Let a program $P$ and an input $I$ be given:
  - $P$ is *logically reactive wrt $I$*: at least one logically coherent global status.
  - $P$ is *logically deterministic wrt $I$*: at most one logically coherent global status.
  - $P$ is *logically correct wrt $I$*: logically reactive and deterministic.
  - $P$ is *logically correct*: logically correct wrt all possible input events.
Logical Correctness

- Pure Esterel programs can be analyzed for logical correctness by exhaustive case analysis.

- Given the status of each input signal, one can make all possible assumptions about the global status and check them individually.

- Logical correctness is decidable ☺ – but NP complete 😞

- Logical correctness can be counter-intuitive – other basis for language semantics needed.
Logical Correctness

```plaintext
module P1:
  input I;
  output O;
  signal S1, S2 in
    present I then emit S1 end
    ||
    present S1 else emit S2 end
    ||
    present S2 then emit O end
  end signal
end module
```

- I present: Assumption S1 present, S2 not present, O not present
  - Justification: The emit S1 statement is executed justifying the assumption S1 present, no emit S2 and emit O statements are executed, justifying the assumption S2 absent and O absent.
- I absent: Assumption S1 absent, S2 present, O present.
  - Justification: The emit S1 statement is not executed justifying the assumption S1 absent, the emit S2 statement is executed justifying the assumption S2 present and the emit O statement is executed justifying the assumption O present.
- All other assumptions can be shown to be logically incoherent.
Logical Correctness

module P2:
  output O;
  present O
  else emit O
  end present
end module

module P3:
  output O;
  present O
  then emit O
  end present
end module

non-reactive

reactive, but non-deterministic
Logical Correctness

module P4:
present O1 then emit O1 end
||
present O1 then
  present O2 else emit O2 end
end

logically correct
Acyclicity and Constructiveness

- Esterel programs can be required to be **acyclic**:
  - No dependency cycles wrt signal dependences
  - Basic idea:
    - present (S) then emit P else emit Q end: $S \rightarrow P$, $S \rightarrow Q$
  - Can be defined precisely and checked at compile time.
  - **BUT**: good programs will be rejected.

- Weaker property called **constructiveness**:
  - Cyclic programs can be constructive
  - Can be checked at compile time
  - More programs will be accepted, but constructiveness is harder to check than acyclicity.
Examples

module P5:
output O;
  present O
    else emit O
  end present
end module

non-reactive

module P6:
output O;
  present O
    then emit O
  end present
end module

reactive, but non-deterministic

Both are rejected by cyclicity test.
Examples

module P7:
input I;
output O1,O2;
present I then
  present O2 then emit O1 end
else
  present O1 then emit O2 end
end present
end module

rejected by acyclicity test
reactive and deterministic

module P7:
output O;
present O then nothing end emit O;
end module

rejected by acyclicity test
(emit O and test in same instant, the test depending on the emit)
Which Semantics to Adopt?

- **Logical correctness** is not in accordance with the **intention** of the language, ie with its **intuitive semantics** and with the **intended sequential character** of test statements.

- Example:

  ```module P10:
  present O then nothing end; emit O```

  is **logically correct**, but the information that O is present flows **backwards** across the sequencing operator ; contradicting the basic intuition about sequential execution.

- Aside from the explicit concurrency || all Esterel statements are **sequential**.
The Constructive Semantics

- Idea: do not check assumptions about signal statuses, but propagate facts about control flow and signal statuses. Self-justification is replaced by fact-to-fact propagation.

- Accounts for programmer's natural way of thinking: in terms of cause and effect.

- Three-valued logic for signals: present +, absent −, unknown ⊥.

- In each instant the statuses of the input signals are given by the environment and the statuses of the other signals are initially set to unknown.
The Constructive Semantics

- Three equivalent presentations:
  - Constructive behavioral semantics
    - Derived from the logical behavioral semantics
    - Constructive restrictions are added to the logical coherence rule
  - Constructive operational semantics
    - Based on term rewriting rules defining microstep sequences
    - Simplest way of defining an efficient interpreter
  - Circuit semantics
    - Translation of program into constructive circuits
    - Core of the Esterel v5 compiler.
Constructive Behavioral Semantics

- Logical coherence semantics augmented by reasoning about what a program must or cannot do, both predicates being disjoint and defined in a constructive way.

- The must predicate determines which signals are present and which statements are executed.
- The cannot predicate determines when signals are absent and it serves in pruning out false execution paths.

- A program is accepted as constructive if and only if fact propagation using the must and cannot predicates suffices in establishing presence or absence of all signals.
Constructive Behavioral Semantics

- **Logical Coherence Law:**
  - A signal $S$ is present in an instant iff an $emit S$ statement is executed in this instant.

- **Constructive Coherence Law:**
  - A signal $S$ is present iff an $emit S$ statement must be executed.
  - A signal $S$ is absent iff an $emit S$ statement cannot be executed.

- The cannot predicate can be derived from the can predicate.
Constructive Behavioral Semantics

- A signal can have three statuses:
  - +: known to be present
  - -: known to be absent
  - ⊥: yet unknown
- must and cannot predicates are defined by structural induction on statements.

\[ p ; q \]

- Must (resp. can) execute \( q \) if \( p \) must (resp. can) terminate

\[ \text{present } S \text{ then } p \text{ else } q \text{ end} \]

- \( S \) known to be present -> Test behaves as \( p \)
- \( S \) known to be absent -> Test behaves as \( q \)
- \( S \) yet unknown -> Test can do whatever \( p \) or \( q \) can do; there is nothing the test must do.
Example

```vhdl
module P1:
  input I;
  output O;
  signal S1, S2 in
    present I then emit S1 end (i1)
    ||
    present S1 else emit S2 end (i2)
    ||
    present S2 then emit O end (i3)
end signal
end module
```

- **If I is present:**
  - i1 must take its *then* branch, *emit S1* and terminate → S1 present
  - i2 must take its (empty) *then* branch and cannot take its *else* branch → *emit S2* cannot be executed, S2 cannot be emitted → S2 absent
  - i3 cannot take its *then* branch → O cannot be emitted and is absent.

- **If I is absent:**
  - i1 cannot take its *then* branch → *emit S1* cannot be executed → S1 absent
  - i2 must take its *then* branch → *emit S2* must be executed → S2 present.
  - i3 must take its *then* branch → *emit O* must be executed → O present.
Example 2 – Part 1

Analyze what signal S must do with status ⊥ for O.

- Analyze body with status ⊥ for O and S.
- S must be emitted.
- Thus: redo the analysis with status ⊥ for O and + for S.
- Status of O is unknown: there is nothing that the present statement must do. Progress can only be made by analyzing what we cannot do in the branches of the test.
- The then branch contains a present S test. Since S is known to be present, we cannot take the implicit else branch. Since the then branch is a pause statement it cannot terminate. Therefore the emit O statement cannot be executed and O cannot be emitted.
- As a consequence O must be set absent and the analysis must be redone with status – for O.
Example 2 – Part 2

- Analyze what signal $S$ must do with status – for $O$.
  - The implicit else branch of the present $O$ test that terminates execution must be taken.
  - The program is *constructive* since we have fully determined the signal statuses.

```vhdl
module P2:
  output O;
  signal S in
      emit S;
      present O then
        present S then
          pause
        end;
      emit O
  end
end signal
```
Constructive Behavioral Semantics

- signal $S$ in $p$ end
  - Can: recursively analyze $p$ with status $\perp$ for $S$
  - Must:
    - Assume we already know that we must execute the declaration in some signal context $E$
    - Must compute final status of $S$ to determine signal context of $p$
    - First analyze $p$ in $E$ augmented by setting the unknown status $\perp$ for $S$
    - If $S$ must be emitted:
      - propagate this information by reanalyzing $p$ in $E$ with $S$ present
      - This may generate more information about the other signals
    - If $S$ cannot be emitted:
      - reanalyze $p$ in $E$ with $S$ absent
Constructive Behavioral Semantics – Formal Definition

- Let $S$ be a set of signals. An event $E$ is a mapping $E : S \rightarrow B_{\perp} = \{+, -, \perp\}$ which assigns a status from $B_{\perp}$ to all signals in $S$.

- Notation:
  - $s^+: E(s) = +$
  - $s^-: E(s) = -$
  - $E \subseteq E': s^+ \text{ in } E \Rightarrow s^+ \text{ in } E'$

- Singleton event $\{s^+\}$:
  - $\{s^+\}(s) = +$ and $\{s^+\}(s') = -$ for all $s' \neq s$

- Let an event $E$ for a set $S$ be given, a signal $s$ possibly not in $S$ and a status $b$ in $B_{\perp}$. Then $E^*s^b$ is an event for the set $S \cup \{s\}$ where $E^*s^b(s) = b$ and $E^*s^b(s') = E(s') \ \forall s' \neq s$. 
Constructive Behavioral Semantics – Formal Definition

- The statements *nothing*, *pause* and *exit* are represented by completion codes $k \geq 0$:
  - *nothing* is encoded by 0
  - *pause* is encoded by 1
  - *exit* $T$ is encoded by 2, if the directly enclosing trap declaration is that of $T$, and $n + 2$ if $n$ trap declarations have to be traversed before reaching that of $T$.

- Each control thread returns a completion code $k \geq 0$ when it has completed its execution in that instant. The completion code is generated by executing a $k$ statement, ie a *nothing*, *pause* or *exit* $T$ kernel statement.
Constructive Behavioral Semantics – Formal Definition

- Given a program $P$ with body $p$ and an input event $I$. A reaction of the program is given by a behavioral transition of the form $P \xrightarrow{O} P'$

  where $O$ is an output event and the resulting program $P'$ is the new state reached by $P$ after the reaction. $P'$ is called the derivative of $P$ by the reaction.

- The statement transition relation has the form $p \xrightarrow{E,k} p'$

  where

  - $E$ is an event that defines the status of all signals in the scope of $p$
  - $E'$ is an event composed of all signals emitted by $p$ in the reaction, $k$ is the completion code returned.

  The statement $p'$ is called the derivative of $p$ by the reaction.

- Given a program $P$ with body $p$ and an input event $I$ then:

  $$P \xrightarrow{O} P' \iff p \xrightarrow{O,k} p' \text{ for some } k$$
Trap Level Propagation

- Let $\tau: \text{Var} \rightarrow \mathbb{N}$ be a function that maps trap names to the trap nesting level of the trap definition.
- $\uparrow$ makes the trap level explicit. We will abbreviate $\uparrow \ldots \uparrow$ by $\uparrow^n$.
- Let $\tilde{E}=(E, \tau)$ be the extended signal environment such that $E$ is an event and $\tau$ a trap nesting level function.

The Must function determines what must be done in a reaction:

$\text{Must}(p, \tilde{E}) = \langle S, K \rangle$ where

- $\tilde{E}$ is an extended signal environment such that initially $\tau=\emptyset$,
- $S$ is the set of signals that $p$ must emit
- $K$ is the set of completion codes that $p$ must return.

We write

$\text{Must}(p, E) = \langle S, K \rangle =: \langle \text{Must}_s(p,E), \text{Must}_k(p,E) \rangle$
Constructive Behavioral Semantics – Formal Definition

- The function $\text{Cannot}^m(p, \bar{E})$ is used to prune out false paths.

\[ \text{Cannot}^m(p, \bar{E}) = \langle \text{Cannot}_s^m(p, \bar{E}), \text{Cannot}_k^m(p, \bar{E}) \rangle = \langle S, K \rangle \]

- $\bar{E}$ is an extended signal environment such that initially $= \emptyset$,
- $S$ is the set of signals that $p$ cannot emit
- $K$ is the set of completion codes that $p$ cannot exit with when the input event is $E$.
- $m \in \{+, \bot\}$ indicates whether it is known that the statement $p$ must be executed in the event $E$. The case $m = -$ will never occur since $\text{Cannot}$ will only be called for potentially executable statements.

- In the following, we will use $\text{Can}^m(p, \bar{E})$ since it is easier to be defined formally; from this, $\text{Cannot}^m(p, \bar{E})$ can be determined by componentwise complementation.
Constructive Behavioral Semantics – Formal Definition

- **Must** and **Can** are defined by structural induction over the kernel statements.

\[
\text{Must}(k, \overline{E}) = \text{Can}^m(k, \overline{E}) = \langle \emptyset, \{k\} \rangle
\]

\[
\text{Must}(\text{emit } S, \overline{E}) = \text{Can}^m(\text{emit } S, \overline{E}) = \langle \{s\}, \{0\} \rangle
\]

\[
\text{Must}(\text{present } s \text{ then } p \text{ else } q \text{ end}, \overline{E}) =
\begin{cases}
\text{Must}(p, \overline{E}), & \text{if } s^+ \in E \\
\text{Must}(q, \overline{E}), & \text{if } s^- \in E \\
\langle \emptyset, \emptyset \rangle, & \text{if } s^\bot \in E
\end{cases}
\]

\[
\text{Can}^m(\text{present } s \text{ then } p \text{ else } q \text{ end}, E) =
\begin{cases}
\text{Can}^m(p, \overline{E}), & \text{if } s^+ \in E \\
\text{Can}^m(q, \overline{E}), & \text{if } s^- \in E \\
\text{Can}^\bot(p, \overline{E}) \cup \text{Can}^\bot(q, \overline{E}), & \text{if } s^\bot \in E
\end{cases}
\]

\[
\text{Must}(\text{suspend } p \text{ when } s, \overline{E}) = \text{Must}(p, \overline{E})
\]

\[
\text{Can}^m(\text{suspend } p \text{ when } s, \overline{E}) = \text{Can}^m(p, \overline{E})
\]
Constructive Behavioral Semantics – Formal Definition

\[
\text{Must}(p;q, \bar{E}) = \begin{cases} \\
\text{Must}(p, \bar{E}), & \text{if } \{0\} \not\in \text{Must}_k(p, \bar{E}) \\
\left\{ \text{Must}_s(p, \bar{E}) \cup \text{Must}_s(q, \bar{E}), \text{Must}_k(q, \bar{E}) \right\}, & \text{if } \{0\} \in \text{Must}_k(p, \bar{E}) \\
\end{cases}
\]

We analyze q only if p must terminate in which case the completion code 0 of p is discarded.
Constructive Behavioral Semantics – Formal Definition

\[
Can^m(p; q, \overline{E}) = \begin{cases} 
Can^m(p, \overline{E}), & \text{if } 0 \notin Can^m_k(p, \overline{E}) \\
\left\langle Can^m_s(p, \overline{E}) \cup Can^m_s(q, \overline{E}), Can^m_k(p, \overline{E}) \setminus \{0\} \cup Can^m_k(q, \overline{E}) \right\rangle \\
& \text{if } 0 \in Can^m_k(p, \overline{E}) \text{ with} \\
& \quad m' = + \text{ if } m = + \text{ and } 0 \in Must^+_k(p, \overline{E}) \\
& \quad \text{or } m' = \bot \text{ otherwise}
\end{cases}
\]

We analyze q with argument \( m' = + \) if \( m = + \) and if \( p \) must terminate, with argument \( m' = \bot \) otherwise.
Constructive Behavioral Semantics – Formal Definition

\[
\begin{align*}
\text{Must}(\text{loop } p \text{ end}, E) &= \text{Must}(p, E) \\
\text{Can}^m(\text{loop } p \text{ end}, E) &= \text{Can}^m(p, E) \\
\text{Must}(p \parallel q, E) &= \left\langle \text{Must}_S(p, E) \cup \text{Must}_S(q, E), \text{Max}(<\text{Must}_k(p, E), \text{Must}_k(q, E)>), \right. \\
\text{Can}^m(p \parallel q, E) &= \left\langle \text{Can}_S^m(p, E) \cup \text{Can}_S^m(q, E), \text{Max}(<\text{Can}_k^m(p, E), \text{Can}_k^m(q, E)>), \right. \\
\text{Max}(K, L) &= \left\{ \begin{array}{ll} \\
\emptyset & \text{if } K = \emptyset \text{ or } L = \emptyset \\
\{ \max \{k, l\} \} & \text{for } k \in K, l \in L \\
\end{array} \right.
\end{align*}
\]

The Max operation e.g. ensures that \(\parallel\) cannot terminate if one of its branches cannot do so.
Trap Level Propagation

\[
\text{Must}(\uparrow^m \text{trap } T \text{ in } p \text{ end}, \overline{E}) = \text{Must}(\{\uparrow^{m+1} p\}^T, \overline{E})
\]

\[
\text{Must}(\{\uparrow^m q\}^T, \overline{E}) = \left\langle \text{Must}_s(\uparrow^m q, \overline{E}'), \downarrow \text{Must}_k(\uparrow^m q, \overline{E}') \right\rangle
\]

where \( \overline{E}' = (E, \tau') \) and \( \tau'(T) = m \)

\[
\text{Must}(\uparrow^m q, \overline{E}) = \left\langle \text{Must}_s(q, \overline{E}), \uparrow^m \text{Must}_k(q, \overline{E}) \right\rangle
\]

\[
\text{Must}(\uparrow^m \text{exit } T, \overline{E}) = \left\langle \emptyset, \text{Must}_k(\uparrow^m \text{exit } T, \overline{E}) \right\rangle
\]

\[
\text{Must}_k(\uparrow^m \text{exit } T, \overline{E}) = 2 + m - \tau(T)
\]

Intuitive completion code rule:

- \( \text{exit } T \) is encoded by 2, if the directly enclosing trap declaration is that of \( T \),
  and \( n+2 \) if \( n \) trap declarations have to be traversed before reaching that of \( T \).

\[
\text{Must}_k(\text{trap } T \text{ in } \text{exit } T \text{ end}, \overline{E}) = \text{Must}_k(\{\uparrow \text{exit } T\}^T, \overline{E})
\]

\[
= \text{Must}_k(\{2+1-1\}^T, \overline{E}) = \text{Must}_k(\{2\}^T, \overline{E}) = \downarrow 2 = 0
\]
Trap Level Propagation

\[ Can^x(\uparrow^m \text{trap } T \text{ in } p \text{ end }, E) = Can^x(\uparrow^m + 1 \{p\}^T, \overline{E}) \]

\[ Can^x(\{\uparrow^m q\}^T, \overline{E}) = \langle Can^x_S(\uparrow^m q, \overline{E'}, \downarrow Can^x_k(\uparrow^m q, \overline{E'}) \rangle \]

where \( \overline{E'} = (E, \tau') \) and \( \tau'(T) = m \)

\[ Can^x(\uparrow^m q, \overline{E}) = \langle Can^x_S(q, \overline{E}), \uparrow^m Can^x_k(q, \overline{E}) \rangle \]

\[ Can^x(\uparrow^m \text{exit } T, \overline{E}) = \langle \emptyset, Can^x_k(\uparrow^m \text{exit } T, \overline{E}) \rangle \]

\[ Can^x_k(\uparrow^m \text{exit } T, \overline{E}) = 2 + m - \tau(T) \]
Constructive Behavioral Semantics – Formal Definition

\[
\text{Must}(\text{trap } T \text{ in } p \text{ end}, E) = \text{Must}(\{\uparrow p\})
\]
\[
\text{Must}(\{q\}, E) = \begin{cases} 
\text{Must}_s(q, E), & \downarrow \text{Must}_k(q, E) 
\end{cases}
\]
\[
\text{Must}(\uparrow q, E) = \begin{cases} 
\text{Must}_s(q, E), & \uparrow \text{Must}_k(q, E) 
\end{cases}
\]
\[
\text{Can}^m(\text{trap } T \text{ in } p \text{ end}, E) = \text{Can}^m(\{\uparrow p\})
\]
\[
\text{Can}^m(\{q\}, E) = \begin{cases} 
\text{Can}^m_s(q, E), & \downarrow \text{Can}^m_k(q, E) 
\end{cases}
\]
\[
\text{Can}^m(\uparrow q, E) = \begin{cases} 
\text{Can}^m_s(q, E), & \uparrow \text{Can}^m_k(q, E) 
\end{cases}
\]

Trap propagation:

\[
\downarrow k = \begin{cases} 
0, & \text{if } k = 0 \text{ or } k = 2 \\
1, & \text{if } k = 1 \\
k - 1, & \text{if } k > 2 
\end{cases}
\]
\[
\uparrow k = \begin{cases} 
k, & \text{if } k = 0 \text{ or } k = 1 \\
k + 1, & \text{if } k > 1 
\end{cases}
\]
Constructive Behavioral Semantics – Formal Definition

\[
\text{Must}(\text{signal } s \text{ in } p, \overline{E}) = \begin{cases} 
\text{Must}(p, \overline{E} \ast s^+) \setminus \{s\}, & \text{if } s \in \text{Must}_s(p, \overline{E} \ast s^\perp) \\
\text{Must}(p, \overline{E} \ast s^-) \setminus \{s\}, & \text{if } s \not\in \text{Can}_s^+(p, \overline{E} \ast s^\perp) \\
\text{Must}(p, \overline{E} \ast s^\perp) \setminus \{s\}, & \text{otherwise}
\end{cases}
\]

\[
\text{Can}^m(\text{signal } s \text{ in } p, \overline{E}) = \begin{cases} 
\text{Can}^+(p, \overline{E} \ast s^+) \setminus \{s\}, & \text{if } m = + \text{ and } s \in \text{Must}_s(p, \overline{E} \ast s^\perp) \\
\text{Can}^m(p, \overline{E} \ast s^-) \setminus \{s\}, & \text{if } s \not\in \text{Can}_s^+(p, \overline{E} \ast s^\perp) \\
\text{Can}^m(p, \overline{E} \ast s^\perp) \setminus \{s\}, & \text{otherwise}
\end{cases}
\]

- We first analyze the body \( p \) with status \( \perp \) for \( s \) with the same \( m \) argument.
- If \( m = + \) and we find that the signal must be emitted we reanalyze \( p \) with status + for \( s \).
- For both \( m = + \) and \( m = \perp \) if the signal cannot be emitted we reanalyze \( p \) with status – and with the same \( m \).
- Otherwise we return the result of the analysis of \( p \) with status \( \perp \) for \( s \).
- Note that the signal status can be set to + only if \( m = + \). This is necessary to avoid speculative computations.
Constructive Behavioral Semantics

- The constructiveness analysis involves many recomputations: Once a signal status has been set, the body of its declaration (the whole program for an output) has to be reanalyzed, this way re-establishing many facts that are already known.

- The goal of the operational and circuit semantics is to avoid recomputing known facts.
Example

module P4:
  input I;
  output O;
  signal S1, S2 in
  present I then emit S1 end
  present S1 then emit S2 end
  present S2 then emit O end
end module

accepted by constructiveness

module P3:
  input I;
  output O1,O2;
  present I then
    present O2 then emit O1 end
  else
    present O1 then emit O2 end
  end present
end module

rejected by acyclicity test
reactive and deterministic
accepted by constructiveness
Advanced Constructiveness

- Preemption statements (abort) behave as tests for the guard in each instant where the guard is active. Their constructiveness test is straightforward.

- Signal expressions:
  - `not e`: straightforward
  - `e1 or e2`: evaluates to true as soon as one of `e1` or `e2` evaluates to true, even if the other one is still unknown.
  - `e1 and e2`: analogous

- The computation of values of valued signals cannot be lazy since the value is known only when all emitters are either executed or discarded (due to signal combination).
  A statement such as `emit S(2)` is handled as `emit S; ?S:=2;` by the constructiveness test.