The 5-Minute Review Session

1. What is the state of an Esterel program?
The 5-Minute Review Session

1. What is the state of an Esterel program? Which implementation alternatives are there to memorize state?
The 5-Minute Review Session

1. What is the *state* of an Esterel program? Which implementation alternatives are there to memorize state?
2. What are implementation alternatives to interface with the environment, *e.g.*, a device that can be on or off?
The 5-Minute Review Session

1. What is the *state* of an Esterel program? Which implementation alternatives are there to memorize state?
2. What are implementation alternatives to interface with the environment, *e.g.*, a device that can be on or off?
3. What is the relationship between *events* and *states*?
The 5-Minute Review Session

1. What is the state of an Esterel program? Which implementation alternatives are there to memorize state?
2. What are implementation alternatives to interface with the environment, e.g., a device that can be on or off?
3. What is the relationship between events and states?
4. What are possible examples for causality problems?
The Constructive Semantics

The 5-Minute Review Session

1. What is the state of an Esterel program? Which implementation alternatives are there to memorize state?
2. What are implementation alternatives to interface with the environment, e.g., a device that can be on or off?
3. What is the relationship between events and states?
4. What are possible examples for causality problems? What is the reason for these problems?
The 5-Minute Review Session

1. What is the state of an Esterel program? Which implementation alternatives are there to memorize state?
2. What are implementation alternatives to interface with the environment, e.g., a device that can be on or off?
3. What is the relationship between events and states?
4. What are possible examples for causality problems? What is the reason for these problems?
5. When is an Esterel program logically reactive?
The 5-Minute Review Session

1. What is the state of an Esterel program? Which implementation alternatives are there to memorize state?

2. What are implementation alternatives to interface with the environment, e.g., a device that can be on or off?

3. What is the relationship between events and states?

4. What are possible examples for causality problems? What is the reason for these problems?

5. When is an Esterel program logically reactive? ... correct?
Overview

The Constructive Semantics

- External Justification vs. Self-Justification
- The Constructive Behavioral Semantics
- The Constructive Operational Semantics
External Justification vs. Self-Justification

- Programming in Esterel:
  - Analyze input events to generate appropriate output signals
  - Use concurrent statements and intermediate local signals to create modular, well-structured programs
External Justification vs. Self-Justification

- Programming in Esterel:
  - Analyze input events to generate appropriate output signals
  - Use concurrent statements and intermediate local signals to create modular, well-structured programs

- Natural way of thinking:
  - Information propagation by *cause* and *effect*

```
  present I then
    emit 0
  end
```
External Justification vs. Self-Justification

```
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 0 end
end signal
end module
```

▶ Is this logically correct?

▶ Yes!

▶ Is this well-behaved wrt information propagation?

▶ Yes!
External Justification vs. Self-Justification

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

▶ Is this logically correct?
  ▶ Yes!
External Justification vs. Self-Justification

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

▶ Is this logically correct?
  ▶ Yes!
▶ Is this well-behaved wrt information propagation?
External Justification vs. Self-Justification

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 0 end
end signal
end module

▶ Is this logically correct?
  ▶ Yes!

▶ Is this well-behaved wrt information propagation?
  ▶ Yes!
External Justification vs. Self-Justification

module P9:
[
    present 01 then emit 01 end
||
    present 01 then
        present 02 else emit 02 end
    end
]

▶ Is this logically correct?
External Justification vs. Self-Justification

module P9:
[
  present 01 then emit 01 end
||
  present 01 then
    present 02 else emit 02 end
  end
]

▶ Is this logically correct?
▶ Yes!
External Justification vs. Self-Justification

```plaintext
module P9:
[
    present 01 then emit 01 end
||
    present 01 then
    present 02 else emit 02 end
end
]
```

- Is this logically correct?  
  - Yes!
- Is this well-behaved wrt information propagation?
External Justification vs. Self-Justification

```plaintext
module P9:
[
    present 01 then emit 01 end
    ||
    present 01 then
        present 02 else emit 02 end
    end
]
```

▶ Is this logically correct?
  ▶ Yes!
▶ Is this well-behaved wrt information propagation?
  ▶ No!
External Justification vs. Self-Justification

module P9:
[
  present 01 then emit 01 end
  ||
  present 01 then
    present 02 else emit 02 end
  end
]

▶ Is this logically correct?
  ▶ Yes!

▶ Is this well-behaved wrt information propagation?
  ▶ No!

▶ Accepting P9 as correct is
  ▶ Logically possible
  ▶ But against (imperative) intention of the language
External Justification vs. Self-Justification

- “present $S$ then $p$ end”:
  - *First* test the status of $S$, *then* execute $p$ if $S$ is present
  - Status of $S$ should not depend on what $p$ *might* do
External Justification vs. Self-Justification

▶ “present $S$ then $p$ end”:
  ▶ *First* test the status of $S$, *then* execute $p$ if $S$ is present
  ▶ Status of $S$ should not depend on what $p$ might do

▶ Synchrony hypothesis:
  ▶ Ordering implicit in the *then* word is not that of time, but that of *sequential causality*
External Justification vs. Self-Justification

- “present $S$ then $p$ end”:
  - *First* test the status of $S$, *then* execute $p$ if $S$ is present
  - Status of $S$ should not depend on what $p$ *might* do

- Synchrony hypothesis:
  - Ordering implicit in the *then* word is not that of time, but that of sequential causality

- Want actual computation:
  - “*Since* $S$ is present, we take the then branch”
External Justification vs. Self-Justification

- “present $S$ then $p$ end”:
  - *First* test the status of $S$, *then* execute $p$ if $S$ is present
  - Status of $S$ should not depend on what $p$ might do

- Synchrony hypothesis:
  - Ordering implicit in the *then* word is not that of time, but that of sequential causality

- Want actual computation:
  - “*Since* $S$ is present, we take the then branch”

- Don’t want speculative computation:
  - “*If* we assume $S$ present, then we take the then branch”
External Justification vs. Self-Justification

- Aside from the explicit concurrency “||”, all Esterel statements are sequential
- Want to preserve this in the semantics
External Justification vs. Self-Justification

- Aside from the explicit concurrency “||”, all Esterel statements are sequential
- Want to preserve this in the semantics

```plaintext
module P10:
  present 0 then
  nothing;
end;
emit 0
```
External Justification vs. Self-Justification

- Aside from the explicit concurrency “||”, all Esterel statements are sequential
- Want to preserve this in the semantics

```plaintext
module P10:
  present 0 then
    nothing;
  end;
  emit 0
```

- This is logically correct
External Justification vs. Self-Justification

- Aside from the explicit concurrency “||”, all Esterel statements are sequential
- Want to preserve this in the semantics

```plaintext
module P10:
  present 0 then
    nothing;
  end;
  emit 0
```

- This is logically correct
- But still want to reject it:
  - In the logical semantics, the information that 0 is present flows backwards across the sequencing operator
  - Contradicts basic intuition about sequential execution
The Constructive Semantics

Constructive semantics:
- Does not check assumptions about signal statuses
- Instead, propagates facts about control flow and signal statuses
The Constructive Semantics

- Constructive semantics:
  - Does not check assumptions about signal statuses
  - Instead, propagates facts about control flow and signal statuses

- Three equivalent presentations:
  1. Constructive behavioral semantics
  2. Constructive operational semantics
  3. Circuit semantics
The Constructive Semantics

1. Constructive behavioral semantics:
   - Derived from the logical behavioral semantics
   - Adds constructive restrictions to logical coherence rule
   - Is the simplest way of defining the language
The Constructive Semantics

1. Constructive behavioral semantics:
   - Derived from the logical behavioral semantics
   - Adds constructive restrictions to logical coherence rule
   - Is the simplest way of defining the language

2. Constructive operational semantics:
   - Based on an interpretation scheme expressed by term rewriting rules defining microstep sequences
   - Is the simplest way of defining an efficient interpreter
1. Constructive behavioral semantics:
   ▶ Derived from the logical behavioral semantics
   ▶ Adds constructive restrictions to logical coherence rule
   ▶ Is the simplest way of defining the language

2. Constructive operational semantics:
   ▶ Based on an interpretation scheme expressed by term rewriting rules defining microstep sequences
   ▶ Is the simplest way of defining an efficient interpreter

3. Circuit semantics:
   ▶ Translation of programs into constructive circuits
   ▶ Is the core of the Esterel v5 compiler
The Constructive Behavioral Semantics

- ...retains the spirit of the logical coherence semantics
- ...adds reasoning about what a program must or cannot do
The Constructive Behavioral Semantics

- ... retains the spirit of the logical coherence semantics
- ... adds reasoning about what a program must or cannot do
- Define disjoint predicates to express
  - “A statement must terminate, must pause, must exit a trap $T$, or must emit a signal $S$”
  - “A statement cannot terminate, cannot pause, cannot exit a trap $T$, or cannot emit a signal $S$”
The Constructive Behavioral Semantics

- ...retains the spirit of the logical coherence semantics
- ...adds reasoning about what a program must or cannot do
- Define disjoint predicates to express
  - “A statement must terminate, must pause, must exit a trap $T$, or must emit a signal $S$”
  - “A statement cannot terminate, cannot pause, cannot exit a trap $T$, or cannot emit a signal $S$”
- The **Must (Cannot)** predicate determines
  - Which signals are present (absent)
  - Which statements are (cannot be) executed
Recall: Logical Coherence Law

A signal $S$ is present in an instant iff an “em$t S$” statement is executed in this instant.
Recall: Logical Coherence Law

A signal $S$ is present in an instant iff an “emit $S$” statement is executed in this instant.

Replace with disjoint Constructive Coherence Laws:

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 Constructive Behavioral Semantics

- Define *Must* and *Cannot* predicates by structural induction on statements.
- A signal can have three statuses:
  - “+”: known to be present
  - “−”: known to be absent
  - ⊥: yet unknown
The Constructive Behavioral Semantics

- Define *Must* and *Cannot* predicates by structural induction on statements.
- A signal can have three statuses:
  - “+”: known to be present
  - “−”: known to be absent
  - “⊥”: yet unknown
- Is technically easier to define the *Cannot* predicate as the negation of a *Can* predicate.
  - No constructiveness problem here as we only deal with finite sets.
The Constructive Behavioral Semantics

\[ p; q \] (Sequence)

- Must (resp. can) execute \( q \) if \( p \) must (resp. can) terminate
The Constructive Behavioral Semantics

\[ p;q \ (\text{Sequence}) \]

- Must (resp. can) execute \( q \) if \( p \) must (resp. can) terminate
- \( \text{present } S \text{ then } p \text{ else } q \text{ end} \) (Test)

- If \( S \) is known to be present:
  - Test behaves as \( p \)
- If \( S \) is known to be absent:
  - Test behaves as \( q \)
- If \( S \) is yet unknown:
  - Test can do whatever \( p \) or \( q \) can do
  - There is nothing the test must do. In particular, it does not even have to do what both \( p \) and \( q \) have to do—this is the essence of disallowing speculative execution.
The Constructive Behavioral Semantics

\[ p; q \text{ (Sequence)} \]

- Must (resp. can) execute \( q \) if \( p \) must (resp. can) terminate
- \( \text{present } S \text{ then } p \text{ else } q \text{ end (Test)} \)
- If \( S \) is known to be present:
  - Test behaves as \( p \)

- If \( S \) is yet unknown:
  - Test can do whatever \( p \) or \( q \) can do
- There is nothing the test must do. In particular, it does not even have to do what both \( p \) and \( q \) have to do—this is the essence of disallowing speculative execution.
The Constructive Behavioral Semantics

\[ p;q \text{ (Sequence)} \]

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

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

- If \( S \) is known to be present:
  - Test behaves as \( p \)

- If \( S \) is known to be absent:
  - Test behaves as \( q \)
The Constructive Behavioral Semantics

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

\[
\begin{aligned}
present &\ S \ then \ p \ else \ q \ end \ (\text{Test}) \\
\end{aligned}
\]
- If \( S \) is known to be present:
  - Test behaves as \( p \)
- If \( S \) is known to be absent:
  - Test behaves as \( q \)
- If \( S \) is yet unknown:
  - Test can do whatever \( p \) or \( q \) can do
  - There is nothing the test must do. In particular, it does not even have to do what both \( p \) and \( q \) have to do—this is the essence of disallowing speculative execution.
The Constructive Behavioral Semantics

\( p ; q \) (Sequence)

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

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

- If \( S \) is known to be present:
  - Test behaves as \( p \)

- If \( S \) is known to be absent:
  - Test behaves as \( q \)

- If \( S \) is yet unknown:
  - Test can do whatever \( p \) or \( q \) can do

- There is nothing the test must do. In particular, it does not even have to do what both \( p \) and \( q \) have to do—this is the essence of disallowing speculative execution.
The Constructive Behavioral Semantics

\( p;q \) (Sequence)

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

\[
\begin{array}{l}
\text{present } S \text{ then } p \text{ else } q \text{ end (Test)} \\
\end{array}
\]

- If \( S \) is known to be present:
  - Test behaves as \( p \)
- If \( S \) is known to be absent:
  - Test behaves as \( q \)
- If \( S \) is yet unknown:
  - Test can do whatever \( p \) or \( q \) can do
  - There is nothing the test must do. In particular, it does not even have to do what both \( p \) and \( q \) have to do—this is the essence of disallowing speculative execution.
The Constructive Behavioral Semantics

- Main novelty is in analysis of output and local signals
- Consider local signal here; output signal is similar

signal $S$ in $p$ end (Local signal)
The Constructive Behavioral Semantics

- Main novelty is in analysis of output and local signals
- Consider local signal here; output signal is similar

signal $S$ in $p$ end (Local signal)

*Can* predicate:
- Recursively analyze $p$ with status $\bot$ for $S$
The Constructive Behavioral Semantics

signal \( S \) in \( p \) end (Local signal)

Must predicate:

- Assume we already know that we must execute signal \( S \) in \( p \) end in some signal context \( E \)
- Must compute final status of \( S \) to determine signal context of \( p \)
The Constructive Behavioral Semantics

signal S in p end (Local signal)

Must predicate:

- Assume we already know that we must execute signal S in p end 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 ⊥ for S
The Constructive Behavioral Semantics

signal $S$ in $p$ end (Local signal)

Must predicate:

- Assume we already know that we must execute signal $S$ in $p$ end 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 $\bot$ 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
The Constructive Behavioral Semantics

signal $S$ in $p$ end (Local signal)

*Must* predicate:

- Assume we already know that we must execute signal $S$ in $p$ end 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 $\bot$ 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
- Similarly, if we find that $S$ cannot be emitted:
  - Reanalyze $p$ in $E$ with $S$ absent
Accepting Programs

In the constructive behavioral semantics, a program is accepted as constructive iff fact propagation using the Must and Can (or Cannot) predicates suffices in establishing presence or absence of all output signals (and we can also compute a derivative—see later).

```plaintext
module P1:
    input I;
    output 0;
    signal S1, S2 in
    present I then emit S1 end
    ||
    present S1 else emit S2 end
    ||
    present S2 then emit 0 end
end signal
end module
```
Accepting Programs

module P2:
signal S in
emit S;
present 0 then
  present S then
  pause
end;
emit 0
end
end signal
Accepting Programs

Can analyze this with just propagating facts
  ▶ No need for speculative computation based on assumptions
  ▶ Our analysis still “looks ahead” to see what must/cannot be done, but always builds on facts established so far, not on speculations

```plaintext
module P2:
signal S in
  emit S;
  present O then
    present S then
      pause
    end;
  emit O
end
end signal
```
Accepting Programs

- Can analyze this with just propagating facts
  - No need for speculative computation based on assumptions
  - Our analysis still “looks ahead” to see what must/cannot be done, but always builds on facts established so far, not on speculations
- However, analysis involves recomputations
  - Avoiding this is goal of operational and circuit semantics!
Rejecting Programs

- If the must and cannot predicates bring no information about the status of some signal:
  - Programs is rejected

```plaintext
module P3:
  output O;
  present O else emit O end
end module

module P4:
  output O;
  present O then emit O end
end module
```
Rejecting Programs

- If the must and cannot predicates bring no information about the status of some signal:
  - Programs is rejected

```plaintext
module P3:
  output O;
  present O else emit O end
end module
```
```plaintext
module P4:
  output O;
  present O then emit O end
end module
```
Rejecting Programs

- If the must and cannot predicates bring no information about the status of some signal:
  - Programs is rejected

```plaintext
module P3:
  output O;
  present O else emit O end
end module

module P4:
  output O;
  present 0 then emit 0 end
end module
```
Rejecting Programs

- Constructiveness $\iff$ logical correctness
- But not vice versa!
Rejecting Programs

- Constructiveness $\implies$ logical correctness
- But not vice versa!

```plaintext
module P9:
[
  present 01 then emit 01 end
  ||
  present 01 then
    present 02 else emit 02 end
  end
]
```

$\triangleright$ Both $O_1$ and $O_2$ can be emitted
$\triangleright$ No signal must be emitted
$\triangleright$ No progress—reject $P_9$!
Rejecting Programs

- Constructiveness $\implies$ logical correctness
- But not vice versa!

```plaintext
module P9:
[ [ present O1 then emit O1 end
|| present O1 then
    present 02 else emit 02 end
   end ]
```

- Both O1 and O2 can be emitted
Rejecting Programs

- Constructiveness $\implies$ logical correctness
- But not vice versa!

```
module P9:
[
    present 01 then emit 01 end
    ||
    present 01 then
    present 02 else emit 02 end
    end
]
```

- Both 01 and 02 can be emitted
- No signal must be emitted
Rejecting Programs

- Constructiveness \(\implies\) logical correctness
- But not vice versa!

```plaintext
module P9:
[ 
  present 01 then emit 01 end
  ||
  present 01 then
    present 02 else emit 02 end
  end
]
```

- Both 01 and 02 can be emitted
- No signal must be emitted
- No progress—**reject P9!**
Rejecting Programs

Consider variant of P2:

```plaintext
module P11:
signal S
  present O then
    emit S;
  present S then
    pause
end;
emit O
end
end signal
```

Are not allowed to speculatively execute branches
Again no progress—reject P11!
Rejecting Programs

Consider variant of P2:

```
module P11:
  signal S
  present 0 then
    emit S;
    present S then
      pause
      end;
    emit 0
  end
end signal
```

▶ Are not allowed to speculatively execute branches
Rejecting Programs

Consider variant of P2:

```plaintext
module P11:
signal S
  present 0 then
    emit S;
  present S then
    pause
  end;
  emit 0
end
end signal
```

- Are not allowed to speculatively execute branches
- Again no progress—*reject P11!*
Rejecting Programs

module P12:
  present 0 then
    emit 0;
  else
    emit 0
end

▶ Must reject P12 as well!
▶ Does an equivalent HW-circuit always stabilize?
  (Will come back to this later . . .)
Rejecting Programs

```
module P12:
present 0 then
  emit 0;
else
  emit 0
end
```

- Must reject P12 as well!
Rejection Programs

- Must reject P12 as well!
- Does an equivalent HW-circuit always stabilize? *(Will come back to this later . . .)*
The *Must*, *Cannot*, and *Can* Functions

- *Must* function determines what must be done in a reaction

\[ P \xrightarrow{O} P' \]
The Must, Cannot, and Can Functions

- **Must** function determines what must be done in a reaction $P \xrightarrow{O} P'$

- Has the form $\text{Must}(p, E) = \langle S, K \rangle$
  - $E$: partial event, associating status in $B_{\perp} = \{+, -, \perp\}$ with each signal
  - $S$: set of signals that $p$ must emit
  - $K$: set of completion codes that $p$ must return
    - Is either empty or a singleton
The **Must**, **Cannot**, and **Can** Functions

- **Must** function determines what must be done in a reaction \( P \xrightarrow{O} P' \)
- Has the form \( \text{Must}(p, E) = \langle S, K \rangle \)
  - \( E \): partial event, associating status in \( B_\perp = \{+, -, \perp\} \) with each signal
  - \( S \): set of signals that \( p \) must emit
  - \( K \): set of completion codes that \( p \) must return
    - Is either empty or a singleton
- Use subscripts to access elements of result pair:
  - \( \text{Must}(p, E) = \langle \text{Must}_s(p, E), \text{Must}_k(p, E) \rangle \)
The Must, Cannot, and Can Functions

- $Cannot^m$ function prunes out false paths
- $Cannot^m(p, E) = \langle Cannot^m_s(p, E), Cannot^m_k(p, E) \rangle = \langle S, K \rangle$
- Extra argument $m \in \{+, \bot\}$ indicates whether it is known that $p$ must be executed in event $E$
- $Can^m(p, E)$ is component-wise complement
Definitions of \textit{Must} and \textit{Can}

- Completion and signal emission:

\begin{align*}
    \text{Must}(k, E) &= \text{Can}^m(k, E) = \langle \emptyset, \{k\} \rangle \\
    \text{Must}(!s, E) &= \text{Can}^m(!s, E) = \langle \{s\}, \{0\} \rangle
\end{align*}
Definitions of \textit{Must} and \textit{Can}

- Completion and signal emission:

  \[
  \text{Must}(k, E) = \text{Can}^m(k, E) = \langle \emptyset, \{k\} \rangle
  \]
  \[
  \text{Must}(!s, E) = \text{Can}^m(!s, E) = \langle \{s\}, \{0\} \rangle
  \]

- Suspension:

  \[
  \text{Must}(s \supset p, E) = \text{Must}(p, E)
  \]
  \[
  \text{Can}^m(s \supset p, E) = \text{Can}^m(p, E)
  \]
Definitions of *Must* and *Can*

- Signal test:
Definitions of *Must* and *Can*

- Signal test:

\[
\text{Must}((s?p, q), E) = \begin{cases} 
\text{Must}(p, E) & \text{if } s^+ \in E \\
\text{Must}(q, E) & \text{if } s^- \in E \\
\langle \emptyset, \emptyset \rangle & \text{if } s^\perp \in E 
\end{cases}
\]

\[
\text{Can}^m((s?p, q), E) = \begin{cases} 
\text{Can}^m(p, E) & \text{if } s^+ \in E \\
\text{Can}^m(q, E) & \text{if } s^- \in E \\
\text{Can}^\perp(p, E) \cup \text{Can}^\perp(q, E) & \text{if } s^\perp \in E 
\end{cases}
\]
Definitions of *Must* and *Can*

▶ Sequencing:

\[
\text{Must} \ (p; q, E) = \begin{cases} \\
\text{Must} \ (p, E) & \text{if } 0 \not\in \text{Must}^k(p, E) \\
\langle \text{Must} \ S(p, E) \cup \text{Must} \ S(q, E), \text{Must}^k(q, E) \rangle & \text{if } 0 \in \text{Must}^k(p, E) 
\end{cases}
\]

\[
\text{Can} \ m(p; q, E) = \begin{cases} \\
\text{Can} \ m(p, E) & \text{if } 0 \not\in \text{Can}^m(p, E) \\
\langle \text{Can} \ mS(p, E) \cup \text{Can} \ m'(S(q, E), \text{Can}^m(p, E) \setminus 0 \cup \text{Can} \ m'(q, E) 
\end{cases}
\]

with \( \text{Can} \ m' = \begin{cases} + & \text{if } m = + \land 0 \in \text{Must}^k(p, E) \\
\bot & \text{otherwise} \end{cases} \)
Definitions of \textit{Must} and \textit{Can}

\begin{itemize}
  \item Sequencing:
\end{itemize}

\[
\text{Must}(p; q, E) = \begin{cases}
  \text{Must}(p, E) & \text{if } 0 \not\in \text{Must}_k(p, E) \\
  \langle \text{Must}_S(p, E) \cup \text{Must}_S(q, E), \text{Must}_k(q, E) \rangle & \text{if } 0 \in \text{Must}_k(p, E)
\end{cases}
\]

\[
\text{Can}^m(p; q, E) = \begin{cases}
  \text{Can}^m(p, E) & \text{if } 0 \not\in \text{Can}^m_k(p, E) \\
  \langle \text{Can}^m_S(p, E) \cup \text{Can}^m_S(q, E), \text{Can}^m_k(p, E) \setminus 0 \cup \text{Can}^m_k(q, E) \rangle & \text{if } 0 \in \text{Can}^m_k(p, E)
\end{cases}
\]

\[
\text{with } m' = \begin{cases}
  + & \text{if } m = + \land 0 \in \text{Must}_k(p, E) \\
  \bot & \text{otherwise}
\end{cases}
\]
Definitions of *Must* and *Can*

- Local signal declaration:
Definitions of \textit{Must} and \textit{Can}

- Local signal declaration:

\[
\text{Must}(p \setminus s, E) = \begin{cases} 
\text{Must}(p, E \ast s^+) \setminus s & \text{if } s \in \text{Must}_S(p, E \ast s^\perp) \\
\text{Must}(p, E \ast s^-) \setminus s & \text{if } s \notin \text{Can}_S^+(p, E \ast s^\perp) \\
\text{Must}(p, E \ast s^\perp) \setminus s & \text{otherwise}
\end{cases}
\]

\[
\text{Can}^m(p \setminus s, E) = \begin{cases} 
\text{Can}^+(p, E \ast s^+) \setminus s & \text{if } m = + \text{ and } s \in \text{Must}_S(p, E \ast s^\perp) \\
\text{Can}^m(p, E \ast s^-) \setminus s & \text{if } s \notin \text{Can}_S^+(p, E \ast s^\perp) \\
\text{Can}^m(p, E \ast s^\perp) \setminus s & \text{otherwise}
\end{cases}
\]
Definitions of *Must* and *Can*

- Note the *Can/Must* asymmetry: in the *Can*-predicate of the local signal declaration, check for $m = +$ before calling *Must* to avoid speculative computation.
- Otherwise, would accept program

```plaintext
present 0 then
  signal S in
    emit S
  ||
  present S else emit 0 end
end
```
Definitions of *Must* and *Can* 

- Loop:

\[
\begin{align*}
\text{Must} & \left( p \ast, E \right) = \text{Must} \left( p, E \right) \\
\text{Can} & \left( m, p \ast, E \right) = \text{Can} \left( m, p, E \right)
\end{align*}
\]

\[
\begin{align*}
\text{Must} & \left( p \mid q, E \right) = \langle \text{Must} \ S(p, E) \cup \text{Must} \ S(q, E), \text{Max} \left( \text{Must} k(p, E), \text{Must} k(q, E) \right) \rangle \\
\text{Can} & \left( m, p \mid q, E \right) = \langle \text{Can} \ S(p, E) \cup \text{Can} \ S(q, E), \text{Max} \left( \text{Can} m k(p, E), \text{Can} m k(q, E) \right) \rangle
\end{align*}
\]

The Max-operator on sets of completion codes is defined as

\[
\text{Max} \left( K, L \right) = \\
\begin{cases} 
\emptyset & \text{if } K = \emptyset \text{ or } L = \emptyset \\
\{ \text{max}(k, l) \mid k \in K, l \in L \} & \text{if } K, L \neq \emptyset
\end{cases}
\]
Definitions of \textit{Must} and \textit{Can}

- Loop:

\[ \text{Must}(p^{*}, E) = \text{Must}(p, E) \]
\[ \text{Can}^m(p^{*}, E) = \text{Can}^m(p, E) \]
Definitions of *Must* and *Can*

- **Loop:**
  
  \[
  \text{Must}(p^*, E) = \text{Must}(p, E) \\
  \text{Can}^m(p^*, E) = \text{Can}^m(p, E)
  \]

- **Parallel:**
Definitions of \textit{Must} and \textit{Can}

- Loop:
  \[ \text{Must}(p^*, E) = \text{Must}(p, E) \]
  \[ \text{Can}^m(p^*, E) = \text{Can}^m(p, E) \]

- Parallel:
  \[ \text{Must}(p|q, E) = \langle \text{Must}_S(p, E) \cup \text{Must}_S(q, E), \]
  \[ \text{Max}(\text{Must}_k(p, E), \text{Must}_k(q, E)) \rangle \]
  \[ \text{Can}^m(p|q, E) = \langle \text{Can}_S^m(p, E) \cup \text{Can}_S^m(q, E), \]
  \[ \text{Max}(\text{Can}_k^m(p, E), \text{Can}_k^m(q, E)) \rangle \]

The \textit{Max}-operator on sets of completion codes is defined as

\[ \text{Max}(K, L) = \begin{cases} 
\emptyset & \text{if } K = \emptyset \text{ or } L = \emptyset \\
\{ \max(k, l) | k \in K, l \in L \} & \text{if } K, L \neq \emptyset 
\end{cases} \]
Definitions of *Must* and *Can*

▶ Trap:
Definitions of *Must* and *Can*

▶ Trap:

\[
\text{Must}(\{p\}, E) = \langle \text{Must}_S(p, E), \downarrow \text{Must}_k(p, E) \rangle
\]

\[
\text{Can}^m(\{p\}, E) = \langle \text{Can}^m_S(p, E), \downarrow \text{Can}^m_k(p, E) \rangle
\]

▶ Shift:
Definitions of *Must* and *Can*

▶ Trap:

\[
\text{Must}(\{ p \}, E) = \langle \text{Must}_S(p, E), \downarrow \text{Must}_k(p, E) \rangle
\]

\[
\text{Can}^m(\{ p \}, E) = \langle \text{Can}_S^m(p, E), \downarrow \text{Can}_k^m(p, E) \rangle
\]

▶ Shift:

\[
\text{Must}(\uparrow p, E) = \langle \text{Must}_S(p, E), \uparrow \text{Must}_k(p, E) \rangle
\]

\[
\text{Can}^m(\uparrow p, E) = \langle \text{Can}_S^m(p, E), \uparrow \text{Can}_k^m(p, E) \rangle
\]
Definition of the Constructive Behavioral Semantics

The constructive behavioral semantics of a given program is defined by a two-step procedure, yielding the current reaction and the derivative:

1. Compute output event $O$ using *Must* and *Cannot* predicates
   - This fails if status of some output signal cannot be determined to be $+$ or $-$
Definition of the Constructive Behavioral Semantics

The constructive behavioral semantics of a given program is defined by a two-step procedure, yielding the current reaction and the derivative:

1. Compute output event $O$ using *Must* and *Cannot* predicates
   - This fails if status of some output signal cannot be determined to be $+$ or $-$

2. Compute behavioral transition yielding program derivative
   - This fails if body of some loop is found to terminate instantaneously
   - This also fails if we cannot establish the presence/absence of a local signal
Definition of the Constructive Semantics

Step 1: Compute output event $O$

Approach:
- Start with undefined $O$ (all output signal statuses $= \bot$)
- Iteratively enrich $O$ using Must and Can information
- Terminate when this stabilizes (guaranteed by monotonicity)

Formalize this as computation of a least fixed point (see draft book)
Algorithm to Compute Outputs

function computeOut(P, I)
    \[ E = I \cup \{s^\perp | s \in \text{Out}(P)\} \]
    do
        \[ E' = E \]
        \[ \text{can} = \text{Can}_S^+(P, E) \]
        \[ \text{must} = \text{Must}_S(P, E) \]
        \[ E = I \cup \{s^+ | s \in \text{must}\} \]
        \[ \cup \{s^- | s \in \text{Out}(P) \setminus \text{can}\} \]
        \[ \cup \{s^\perp | s \in \text{can} \setminus \text{must}\} \]
    while (\[ E' \neq E \])
    if \[ \exists s : s^\perp \in E \] then error ("not constructive")
    return E
Example for \textit{Can} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$. 
Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S\perp, O\perp\}$. 
Example for *Can* analysis

Consider the program \( p = !S; S?!O, 1 \) and environment \( \{S\|, O\|\} \).

\[
Can^+ (!S, \{S\|, O\|\}) =
\]
Example for \textit{Can} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S\perp, O\perp\}$.

\[
\text{Can}^+(!S, \{S\perp, O\perp\}) = \langle\{S\}, \{0\}\rangle
\]
Example for *Can* analysis

Consider the program $p = \mathit{!}S; S?\mathit{!}O, 1$ and environment $\{S, O\}$. 

\[
\begin{align*}
\mathit{Can}^+ (\mathit{!}S, \{S, O\}) &= \langle \{S\}, \{0\} \rangle \\
\mathit{Must}_k (\mathit{!}S, \{S, O\}) &=
\end{align*}
\]
Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

\[
\text{Can}^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle
\]

\[
\text{Must}_k(!S, \{S^\perp, O^\perp\}) = \{0\}
\]
Example for *Can* analysis

Consider the program \( p = !S; S?!O, 1 \) and environment \( \{S \perp, O \perp\} \).

\[
\begin{align*}
Can^+(!S, \{S \perp, O \perp\}) & = \langle \{S\}, \{0\} \rangle \\
Must_k(!S, \{S \perp, O \perp\}) & = \{0\} \\
Can^\perp(!O, \{S \perp, O \perp\}) & = \\
\end{align*}
\]
Example for *Can* analysis

Consider the program \( p = \!S; S?!O, 1 \) and environment \( \{S\bot, O\bot\} \).

\[
\begin{align*}
Can^+(\!S, \{S\bot, O\bot\}) &= \langle \{S\}, \{0\} \rangle \\
Must_k(\!S, \{S\bot, O\bot\}) &= \{0\} \\
Can^\bot(\!O, \{S\bot, O\bot\}) &= \langle \{O\}, \{0\} \rangle
\end{align*}
\]
Example for \textit{Can} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S \perp, O \perp\}$.

\[
\begin{align*}
\text{Can}^+ (\!S, \{S \perp, O \perp\}) &= \langle \{S\}, \{0\} \rangle \\
\text{Must}^*_k (\!S, \{S \perp, O \perp\}) &= \{0\} \\
\text{Can}^\perp (\!O, \{S \perp, O \perp\}) &= \langle \{O\}, \{0\} \rangle \\
\text{Can}^\perp (1, \{S \perp, O \perp\}) &=
\end{align*}
\]
Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S\perp, O\perp\}$.

\[
\begin{align*}
\text{Can}^+ ( !S, \{ S\perp, O\perp \} ) &= \langle \{ S \}, \{ 0 \} \rangle \\
\text{Must}_k ( !S, \{ S\perp, O\perp \} ) &= \{ 0 \} \\
\text{Can}^\perp ( !O, \{ S\perp, O\perp \} ) &= \langle \{ O \}, \{ 0 \} \rangle \\
\text{Can}^\perp ( 1, \{ S\perp, O\perp \} ) &= \langle \emptyset, \{ 1 \} \rangle
\end{align*}
\]
Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

\[
\begin{align*}
  Can^+(!S, \{S^\perp, O^\perp\}) &= \langle\{S\}, \{0\}\rangle \\
  Must_k(!S, \{S^\perp, O^\perp\}) &= \{0\} \\
  Can^-(!O, \{S^\perp, O^\perp\}) &= \langle\{O\}, \{0\}\rangle \\
  Can^-(1, \{S^\perp, O^\perp\}) &= \langle\emptyset, \{1\}\rangle \\
  Can^+(S?!O, 1, \{S^\perp, O^\perp\}) &= \langle\emptyset, \{1\}\rangle 
\end{align*}
\]
Example for *Can* analysis

Consider the program \( p = !S; S?!O, 1 \) and environment \( \{ S \perp, O \perp \} \).

\[
\begin{align*}
Can^+(!S, \{ S \perp, O \perp \}) &= \langle \{ S \}, \{ 0 \} \rangle \\
Must_k(!S, \{ S \perp, O \perp \}) &= \{ 0 \} \\
Can^\perp(!O, \{ S \perp, O \perp \}) &= \langle \{ O \}, \{ 0 \} \rangle \\
Can^\perp(1, \{ S \perp, O \perp \}) &= \langle \emptyset, \{ 1 \} \rangle \\
Can^+(S?!O, 1, \{ S \perp, O \perp \}) &= \langle \{ O \}, \{ 0, 1 \} \rangle
\end{align*}
\]
Example for *Can* analysis

Consider the program \( p = !S; S?!O, 1 \) and environment \( \{S \perp, O \perp\} \).

\[
\begin{align*}
Can^+(!S, \{S \perp, O \perp\}) &= \langle\{S\}, \{0\}\rangle \\
Must_k(!S, \{S \perp, O \perp\}) &= \{0\} \\
Can^\perp(!O, \{S \perp, O \perp\}) &= \langle\{O\}, \{0\}\rangle \\
Can^\perp(1, \{S \perp, O \perp\}) &= \langle\emptyset, \{1\}\rangle \\
Can^+ (S?!O, 1, \{S \perp, O \perp\}) &= \langle\{O\}, \{0, 1\}\rangle \\
Can^+ (!S; S?!O, 1, \{S \perp, O \perp\}) &= \langle\{S, O\}, \{0, 1\}\rangle
\end{align*}
\]
Example for *Can* analysis

Consider the program \( p = !S; S?!O, 1 \) and environment \( \{S^\perp, O^\perp\} \).

\[
Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle
\]

\[
Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}
\]

\[
Can^+(O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle
\]

\[
Can^+(1, \{S^\perp, O^\perp\}) = \langle \emptyset, \{1\} \rangle
\]

\[
Can^+(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0, 1\} \rangle
\]

\[
Can^+(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S, O\}, \{0, 1\} \rangle
\]
Example for \textit{Can} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S \perp, O \perp\}$.

\[
\begin{align*}
\text{Can}^+(!S, \{S \perp, O \perp\}) &= \langle \{S\}, \{0\} \rangle \\
\text{Must}_k(!S, \{S \perp, O \perp\}) &= \{0\} \\
\text{Can}^-(!O, \{S \perp, O \perp\}) &= \langle \{O\}, \{0\} \rangle \\
\text{Can}^-(1, \{S \perp, O \perp\}) &= \langle \emptyset, \{1\} \rangle \\
\text{Can}^+(S?!O, 1, \{S \perp, O \perp\}) &= \langle \{O\}, \{0, 1\} \rangle \\
\text{Can}^+(!S; S?!O, 1, \{S \perp, O \perp\}) &= \langle \{S, O\}, \{0, 1\} \rangle
\end{align*}
\]

Gives no new information on signal status
Example for *Must* analysis

Consider the program $p =!S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$
Example for \textit{Must} analysis

Consider the program \( p = !S; S?!O, 1 \) and environment \( \{ S \perp, O \perp \} \)

1. \( \text{Must}(!S, \{ S \perp, O \perp \}) = \)
Example for *Must* analysis

Consider the program $p = !S; S?O, 1$ and environment $\{S^\perp, O^\perp\}$

1. $\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle\{S\}, \{0\}\rangle$
Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{ S^{\perp}, O^{\perp} \}$

1. $\text{Must}(!S, \{ S^{\perp}, O^{\perp} \}) = \langle \{ S \}, \{ 0 \} \rangle$

$\text{Must}(S?!O, 1, \{ S^{\perp}, O^{\perp} \}) = \langle \{ \}, \{ \} \rangle$
Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\bot, O^\bot\}$

1. $Must(!S, \{S^\bot, O^\bot\}) = \langle\{S\}, \{0\}\rangle$
   $Must(S?!O, 1, \{S^\bot, O^\bot\}) = \langle\emptyset, \emptyset\rangle$
Example for \textit{Must} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. \begin{align*}
\text{Must}(!S, \{S^\perp, O^\perp\}) &= \langle\{S\}, \{0\}\rangle \\
\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) &= \langle\emptyset, \emptyset\rangle \\
\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) &= \langle\emptyset, \emptyset\rangle \\
\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) &= \langle\emptyset, \emptyset\rangle
\end{align*}
Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S\bot, O\bot\}$

1. $\text{Must}(!S, \{S\bot, O\bot\}) = \langle\{S\}, \{0\}\rangle$
   $\text{Must}(S?!O, 1, \{S\bot, O\bot\}) = \langle\emptyset, \emptyset\rangle$
   $\text{Must}(!S; S?!O, 1, \{S\bot, O\bot\}) = \langle\{S\}, \emptyset\rangle$
Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. \[ \text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle \]
   \[ \text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle \]
   \[ \text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle \]

2. Update environment to $\{S^+, O^\perp\}$
Example for Must analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. $\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle\{S\}, \{0\}\rangle$
   
   $\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle\emptyset, \emptyset\rangle$
   
   $\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle\{S\}, \emptyset\rangle$

2. Update environment to $\{S^+, O^\perp\}$

3. $\text{Must}(!S, \{S^+, O^\perp\}) =$
Example for \textit{Must} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\bot, O^\bot\}$

1. $\text{Must}(!S, \{S^\bot, O^\bot\}) = \langle\{S\}, \{0\}\rangle$
   
   $\text{Must}(S?!O, 1, \{S^\bot, O^\bot\}) = \langle\emptyset, \emptyset\rangle$

   $\text{Must}(!S; S?!O, 1, \{S^\bot, O^\bot\}) = \langle\{S\}, \emptyset\rangle$

2. Update environment to $\{S^+, O^\bot\}$
3. $\text{Must}(!S, \{S^+, O^\bot\}) = \langle\{S\}, \{0\}\rangle$
Example for \textit{Must} analysis

Consider the program $p = !S; S?O, 1$ and environment $\{S^\bot, O^\bot\}$

1. $\text{Must}(!S, \{S^\bot, O^\bot\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(S?O, 1, \{S^\bot, O^\bot\}) = \langle \emptyset, \emptyset \rangle$
   $\text{Must}(!S; S?O, 1, \{S^\bot, O^\bot\}) = \langle \{S\}, \emptyset \rangle$

2. Update environment to $\{S^+, O^\bot\}$
3. $\text{Must}(!S, \{S^+, O^\bot\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(!O, \{S^+, O^\bot\}) =$


Example for \textit{Must} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S \perp, O \perp\}$

1. $\text{Must}(!S, \{S \perp, O \perp\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(S?!O, 1, \{S \perp, O \perp\}) = \langle \emptyset, \emptyset \rangle$
   $\text{Must}(!S; S?!O, 1, \{S \perp, O \perp\}) = \langle \{S\}, \emptyset \rangle$

2. Update environment to $\{S^+, O \perp\}$
3. $\text{Must}(!S, \{S^+, O \perp\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(!O, \{S^+, O \perp\}) = \langle \{O\}, \{0\} \rangle$
Example for *Must* analysis

Consider the program \( p = !S; S?!O, 1 \) and environment \( \{ S^\perp, O^\perp \} \)

1. \[
    \text{Must}(!S, \{ S^\perp, O^\perp \}) = \langle \{ S \}, \{ 0 \} \rangle \\
    \text{Must}(S?!O, 1, \{ S^\perp, O^\perp \}) = \langle \emptyset, \emptyset \rangle \\
    \text{Must}(!S; S?!O, 1, \{ S^\perp, O^\perp \}) = \langle \{ S \}, \emptyset \rangle
\]

2. Update environment to \( \{ S^+, O^\perp \} \)

3. \[
    \text{Must}(!S, \{ S^+, O^\perp \}) = \langle \{ S \}, \{ 0 \} \rangle \\
    \text{Must}(!O, \{ S^+, O^\perp \}) = \langle \{ O \}, \{ 0 \} \rangle \\
    \text{Must}(S?!O, 1, \{ S^+, O^\perp \}) = \langle \{ S, O \}, \{ 0 \} \rangle
\]
Example for \textit{Must} analysis

Consider the program $p = !S; S?;!O, 1$ and environment $\{S\perp, O\perp\}$

1. $\text{Must}(!S, \{S\perp, O\perp\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(S?;!O, 1, \{S\perp, O\perp\}) = \langle \emptyset, \emptyset \rangle$
   $\text{Must}(!S; S?;!O, 1, \{S\perp, O\perp\}) = \langle \{S\}, \emptyset \rangle$

2. Update environment to $\{S^+, O\perp\}$

3. $\text{Must}(!S, \{S^+, O\perp\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(!O, \{S^+, O\perp\}) = \langle \{O\}, \{0\} \rangle$
   $\text{Must}(S?;!O, 1, \{S^+, O\perp\}) = \langle \{O\}, \{0\} \rangle$
Example for \textbf{Must} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. \quad \text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle
   \quad \text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle
   \quad \text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle

2. Update environment to $\{S^+, O^\perp\}$

3. \quad \text{Must}(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle
   \quad \text{Must}(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle
   \quad \text{Must}(S?!O, 1, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle
   \quad \text{Must}(!S; S?!O, 1, \{S^+, O^\perp\}) = \langle \{S^+, O^\perp\} \rangle
Example for *Must* analysis

Consider the program $p = S; S!O, 1$ and environment $\{S^\bot, O^\bot\}$

1. $\text{Must}(S, \{S^\bot, O^\bot\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(S!O, 1, \{S^\bot, O^\bot\}) = \langle \emptyset, \emptyset \rangle$
   $\text{Must}(S; S!O, 1, \{S^\bot, O^\bot\}) = \langle \{S\}, \emptyset \rangle$

2. Update environment to $\{S^+, O^\bot\}$

3. $\text{Must}(S, \{S^+, O^\bot\}) = \langle \{S\}, \{0\} \rangle$
   $\text{Must}(O, \{S^+, O^\bot\}) = \langle \{O\}, \{0\} \rangle$
   $\text{Must}(S!O, 1, \{S^+, O^\bot\}) = \langle \{O\}, \{0\} \rangle$
   $\text{Must}(S; S!O, 1, \{S^+, O^\bot\}) = \langle \{S, O\}, \{0\} \rangle$
Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S\perp, O\perp\}$

1. $\text{Must}(!S, \{S\perp, O\perp\}) = \langle\{S\}, \{0\}\rangle$
   
   $\text{Must}(S?!O, 1, \{S\perp, O\perp\}) = \langle\emptyset, \emptyset\rangle$

   $\text{Must}(!S; S?!O, 1, \{S\perp, O\perp\}) = \langle\{S\}, \emptyset\rangle$

2. Update environment to $\{S^+, O\perp\}$

3. $\text{Must}(!S, \{S^+, O\perp\}) = \langle\{S\}, \{0\}\rangle$

   $\text{Must}(!O, \{S^+, O\perp\}) = \langle\{O\}, \{0\}\rangle$

   $\text{Must}(S?!O, 1, \{S^+, O\perp\}) = \langle\{O\}, \{0\}\rangle$

   $\text{Must}(!S; S?!O, 1, \{S^+, O\perp\}) = \langle\{S, O\}, \{0\}\rangle$

4. Update environment to $\{S^+, O^+\}$
Example for \textit{Must} analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
   $Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$
   $Must(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$

2. Update environment to $\{S^+, O^\perp\}$
3. $Must(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
   $Must(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$
   $Must(S?!O, 1, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$
   $Must(!S; S?!O, 1, \{S^+, O^\perp\}) = \langle \{S, O\}, \{0\} \rangle$

4. Update environment to $\{S^+, O^+\}$
5. All signals have a defined status $\rightarrow$ done
Definition of the Constructive Semantics

Step 2: Compute transition
Rules are exactly as for logical behavioral semantics—except for changed rules for local signals

\[
\begin{align*}
    p \xrightarrow{E', s^+, k} p' & \quad \text{S}(E') = S(E) \setminus s \\
    \frac{E \xrightarrow{s^+}}{E' s^+} & \\
    p \setminus s \xrightarrow{E', k} p' \setminus s
\end{align*}
\]
Definition of the Constructive Semantics

Step 2: Compute transition
Rules are exactly as for logical behavioral semantics—except for changed rules for local signals

\[
p \xrightarrow{E' * s^+, k} p' \quad S(E') = S(E) \setminus s
\]

\[
p \xrightarrow{E * s^+} p' \quad S(E') = S(E) \setminus s
\]

\[
p \setminus s \xrightarrow{E' , k} p' \setminus s
\]

\[
s \in \text{Must}_s(p, E * s^\perp) \quad p \xrightarrow{E' * s^+, k} p' \quad S(E') = S(E) \setminus s
\]

\[
p \setminus s \xrightarrow{E' , k} p' \setminus s
\]

\[
(csig +)
\]
Definition of the Constructive Semantics

\[
p \xrightarrow{E' * s^-,k} p' \quad S(E') = S(E) \setminus s
\]
\[
p \xrightarrow{E \setminus s} p' \setminus s
\]

is replaced with

\[
s \in \text{Cannot}^+_s(p, E \setminus s^\perp) \quad p \xrightarrow{E' * s^-,k} p' \quad S(E') = S(E) \setminus s
\]
\[
p \xrightarrow{E \setminus s} p' \setminus s
\]

\text{(sig -)}

\text{(csig -)}
The Constructive Operational Semantics

- ... is defined by a rewriting-based interpretation scheme
The Constructive Operational Semantics

- ...is defined by a rewriting-based interpretation scheme
  - Instead of reasoning about what we must do, just do it
The Constructive Operational Semantics

- ...is defined by a rewriting-based interpretation scheme
  - 😊 Instead of reasoning about what we must do, just do it
  - 😞 Formal definition and technical treatment of the constructive *operational* semantics is much heavier than that of the constructive *behavioral* semantics

- Will still take constructive *behavioral* semantics as the primary semantics
The Constructive Operational Semantics

- Decorate signal declarations with status $+, -, \bot$
- Initially, all signals except inputs unknown
- Constructive operational semantics is a micro-step semantics
  - Current state indicated by $\bullet$
The Constructive Operational Semantics

Consider P1 with I present:

```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 0 end
end signal
end module
```
The Constructive Operational Semantics

Fork of the parallel statement:

```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
```
The Constructive Operational Semantics

Only first thread can continue:

```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 0 end 
  end signal 
end module
```
The Constructive Operational Semantics

Now emit S1:

```plaintext
module P1:
  input I⁺;
  output 0⊥;
  signal S1⁺, S2⊥ in
    present I then emit S1 end•
  •||
  • present S1 else emit S2 end
  •||
  • present S2 then emit 0 end
end signal
end module
```
Now the 2nd branch can continue:

```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 0 end
end signal
end module
```
The Constructive Operational Semantics

Cannot emit \( S_2 \) any more:

```
module P1:
  input I^+;
  output O⊥;
  signal S_1^+, S_2^- in
    present I then emit S_1 end\bullet
  ||
    present S_1 else emit S_2 end\bullet
  ||
    \bullet present S_2 then emit 0 end
end signal
end module
```
Now 3rd branch can continue:

```plaintext
module P1:
    input I⁺;
    output O⊥;
    signal S1⁺, S2⁻ in
        present I then emit S1 end
        present S1 then emit 0 end
    end signal
end module
```
Cannot emit 0 any more:

```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 0 end•
    end signal
end module
```
The Constructive Operational Semantics

Synchronize the terminated threads:

```
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
```
Now consider P2:

```plaintext
module P2:
  output O ⊥;
  signal S ⊥ in
    emit S;
    present O then
      present S then
        pause
      end;
      emit O
    end
  emit O
end
```

The Constructive Operational Semantics
The Constructive Operational Semantics

After 3 microsteps:

```plaintext
module P2:
  output O ⊥;
  signal S+ in
  emit S;
  • present O then
    present S then
    pause
  end;
  emit 0
end
end signal
end module
```
The Constructive Operational Semantics

Perform cannot analysis (as in constructive behavioral semantics)—and set 0 absent:

```
module P2:
output 0⁻;
signal S⁺ in
  emit S;
  ⋄ present 0 then
    present S then
    pause
  end;
  emit 0
end
end signal
end module
```
The Constructive Operational Semantics

Take implicit else branch of test:

```plaintext
module P2:
  output $0^{-}$;
  signal $S^{+}$ in
    emit $S$;
    present $\bar{O}$ then
      present $S$ then
        pause
      end;
    emit $0$
  end
end signal
end module
```
The Constructive Operational Semantics

- Statuses evolve monotonically
  - Hence avoid most of the recomputations that take place in the constructive behavioral semantics
- Rejecting programs is similar to constructive behavioral semantics
The Constructive Operational Semantics

- Statuses evolve monotonically
  - Hence avoid most of the recomputations that take place in the constructive behavioral semantics
- Rejecting programs is similar to constructive behavioral semantics

```plaintext
module P3:
  output O;
  present O else emit O end
end module
```
The Constructive Operational Semantics

- Statuses evolve monotonically
  - Hence avoid most of the recomputations that take place in the constructive behavioral semantics
- Rejecting programs is similar to constructive behavioral semantics

```
module P3:
  output 0;
  present 0 else emit 0 end
end module
```

- No possible initial microstep $\nrightarrow$ cannot set $O^+$
The Constructive Operational Semantics

- Statuses evolve monotonically
  - Hence avoid most of the recomputations that take place in the constructive behavioral semantics
- Rejecting programs is similar to constructive behavioral semantics

```plaintext
module P3:
output 0;
present 0 else emit 0 end
end module
```

- No possible initial microstep $\implies$ cannot set $O^+$
- Potential path to emit 0 $\implies$ cannot set $O^-$
Summary of Constructive Interpretation

Signals:
- Signals are shared objects with status \{+,-,\perp\}
Summary of Constructive Interpretation

Signals:
- Signals are shared objects with status \{+, −, ⊥\}
- Signal status initialization:
  - Input signals are initialized according to the input event
  - Other signals initialized to ⊥
Summary of Constructive Interpretation

Signals:
- Signals are shared objects with status \{+,-,\bot\}
- Signal status initialization:
  - Input signals are initialized according to the input event
  - Other signals initialized to \bot
- Signal status changes:
  - Status of a signal $S$ changes from $\bot$ to $+$ as soon as an “emit $S$” statement is executed
  - Status of a signal $S$ changes from $\bot$ to $-$ as soon as all the “emit $S$” statements have been found unreachable by the cannot false path analysis
Summary of Constructive Interpretation

Control:

- Sequential threads of control forked by parallel statements
Summary of Constructive Interpretation

Control:

- Sequential threads of control forked by parallel statements
- When a thread reaches a “present $S$” statement:
  - As long as the status of $S$ is $\perp$:
    - Control remains there, frozen,
  - As soon as $S$ has a non-$\perp$ status:
    - Control can resume
Summary of Constructive Interpretation

Control:

► Sequential threads of control forked by parallel statements
► When a thread reaches a “present \( S \)” statement:
  ► As long as the status of \( S \) is \( \perp \):
    ► Control remains there, frozen,
  ► As soon as \( S \) has a non-\( \perp \) status:
    ► Control can resume
► If several threads are enabled, any one of them can be chosen
Summary of Constructive Interpretation

Control:

▶ Threads are stopped by termination or by executing pause or exit statements

▶ Parallel statements synchronize stopped threads, as explained in the intuitive semantics

▶ Finally, the false path analysis explores all possible instantaneous paths towards emit statements
  ▶ Takes into account all facts established so far
  ▶ No speculative reasoning
Summary of Constructive Interpretation

Program Acceptance:

- Given an input, a program is accepted if the analysis succeeds in setting each signal status to a defined value $+$ or $-$.
- Logical correctness is guaranteed for accepted programs.
To Go Further