Semantics of nondeterminism

Paul Blain Levy

University of Birmingham

November 17, 2009
1. Denotational Semantics
2. Nondeterminism
How do we describe the meaning of a programming language?

One approach is to say how to run a program (e.g. with an interpreter). This is called an operational semantics.

Denotational semantics gives a denotation for every piece of code—even if it's not a complete program. If \( M \) is a piece of code, we write \( \llbracket M \rrbracket \) for its denotation.

Compositionality If a big piece of code is made up from some components, the meaning of the big piece must be given in terms of the meaning of the components.

A denotational semantics has to be proven to agree with the operational semantics—otherwise it's useless.
How do we describe the meaning of a programming language?

- One approach is to say how to run a program (e.g. with an interpreter). This is called an operational semantics.
Operational vs Denotational Semantics

How do we describe the meaning of a programming language?

- One approach is to say how to run a program (e.g. with an interpreter). This is called an operational semantics.
- Denotational semantics gives a denotation for every piece of code—even if it’s not a complete program.
Operational vs Denotational Semantics

How do we describe the meaning of a programming language?

- One approach is to say how to run a program (e.g. with an interpreter). This is called an operational semantics.

- Denotational semantics gives a denotation for every piece of code—even if it’s not a complete program.

- If $M$ is a piece of code, we write $\llbracket M \rrbracket$ for its denotation.
How do we describe the meaning of a programming language?

- One approach is to say how to run a program (e.g. with an interpreter). This is called an operational semantics.
- **Denotational semantics** gives a denotation for every piece of code—even if it’s not a complete program.
- If $M$ is a piece of code, we write $\llbracket M \rrbracket$ for its denotation.
- **Compositionality** If a big piece of code is made up from some components, the meaning of the big piece must be given in terms of the meaning of the components.
How do we describe the meaning of a programming language?

- One approach is to say how to run a program (e.g. with an interpreter). This is called an operational semantics.

- Denotational semantics gives a denotation for every piece of code—even if it’s not a complete program.

- If $M$ is a piece of code, we write $\llbracket M \rrbracket$ for its denotation.

- Compositionality If a big piece of code is made up from some components, the meaning of the big piece must be given in terms of the meaning of the components.

- A denotational semantics has to be proven to agree with the operational semantics—otherwise it’s useless.
Example: simple \texttt{while} language

Our language has two (nonnegative) integer variables \(x\) and \(y\).
Example: simple \texttt{while} language

Our language has two (nonnegative) integer variables $x$ and $y$. Integer expressions are given by the BNF grammar

\[
E ::= x \mid y \mid E + E \mid E \ast E \mid n \quad (n \in \mathbb{N})
\]
Example: simple while language

Our language has two (nonnegative) integer variables $x$ and $y$. Integer expressions are given by the BNF grammar

$$E ::= x \mid y \mid E + E \mid E \ast E \mid n \quad (n \in \mathbb{N})$$

Boolean expressions are given by the grammar

$$B ::= E > E \mid E = E \mid \text{true} \mid \text{not } B \mid B \text{ and } B$$
Example: simple while language

Our language has two (nonnegative) integer variables $x$ and $y$. Integer expressions are given by the BNF grammar

$$
E ::= x \mid y \mid E + E \mid E \times E \mid n \quad (n \in \mathbb{N})
$$

Boolean expressions are given by the grammar

$$B ::= E > E \mid E = E \mid \text{true} \mid \text{not } B \mid B \text{ and } B$$

Commands are given by the grammar

$$M ::= \text{skip} \mid x := E \mid y := E$$

$$M; M \mid \text{if } B \text{ then } M \text{ else } M \mid \text{while } B \text{ do } M$$
A state is a pair of integers e.g. (4, 17).
This means that currently $x = 4$ and $y = 17$. 
A state is a pair of integers e.g. \((4, 17)\).
This means that currently \(x = 4\) and \(y = 17\).

The set of states is \(S \overset{\text{def}}{=} \mathbb{N} \times \mathbb{N}\).
A state is a pair of integers e.g. (4, 17). This means that currently $x = 4$ and $y = 17$.

The set of states is $S \overset{\text{def}}{=} \mathbb{N} \times \mathbb{N}$.

Each integer expression $E$ denotes a function $[E]$ from $S$ to $\mathbb{N}$. 
A state is a pair of integers e.g. (4, 17). This means that currently $x = 4$ and $y = 17$.

The set of states is $S \overset{\text{def}}{=} \mathbb{N} \times \mathbb{N}$.

Each integer expression $E$ denotes a function $[E]$ from $S$ to $\mathbb{N}$.

**Example: the meaning of +**

$[E + E']$ is the function mapping a state $s$ to the integer $[E]s + [E']s$. 
A state is a pair of integers e.g. \((4, 17)\).
This means that currently \(x = 4\) and \(y = 17\).

The set of states is \(S \overset{\text{def}}{=} \mathbb{N} \times \mathbb{N}\).

Each integer expression \(E\) denotes a function \([E]\) from \(S\) to \(\mathbb{N}\).

**Example: the meaning of +**
\([E + E']\) is the function mapping a state \(s\) to the integer \([E]s + [E']s\).

Each boolean expression \(B\) denotes a function \([B]\) from \(S\) to \(\mathbb{B}\) (the set of booleans).
If we run a program in a given starting state $s$, there are two possible behaviours:

- it can terminate in another state $s'$
- it can diverge (run silently forever).
Semantics of Commands

If we run a program in a given starting state $s$, there are two possible behaviours:

- it can terminate in another state $s'$
- it can diverge (run silently forever).

We write $S_{\perp}$ for the set of states extended with an extra element $\perp$, representing divergence.

A command $M$ denotes a function $[M]$ from $S$ to $S_{\perp}$. 
Semantics of Commands

If we run a program in a given starting state $s$, there are two possible behaviours:

- it can terminate in another state $s'$
- it can diverge (run silently forever).

We write $S_\perp$ for the set of states extended with an extra element $\perp$, representing divergence.

A command $M$ denotes a function $\llbracket M \rrbracket$ from $S$ to $S_\perp$.

For example, we want the denotation of

\[
x := x + 4;
\]

\[
\text{while } (x > y) \text{ do } \{ x := x + 1 \}
\]

to be the function that maps the state $(x, y)$ to

- $\perp$ if $x + 4 > y$
- $(x + 4, y)$ if $x + 4 \leq y$. 
Example: the meaning of **while**

\[[\text{while } B \text{ do } M] \] is the function mapping a state \( s \) to

- a state \( s' \) if there is a sequence of states \( s = s_0, s_1, \ldots, s_n = s' \) such that
  \[
  [B]s_i = \text{true} \quad \text{and} \quad [M]s_i = s_{i+1} \quad \text{for each } i < n \\
  [B]s_n = \text{false}
  \]

- \( \bot \) if there is a sequence of states \( s = s_0, s_1, \ldots, s_n \) such that
  \[
  [B]s_i = \text{true} \quad \text{and} \quad [M]s_i = s_{i+1} \quad \text{for each } i < n \\
  [B]s_n = \text{true} \quad \text{and} \quad [M]s_n = \bot
  \]

- \( \bot \) if there is an infinite sequence of states \( s = s_0, s_1, \ldots \) such that
  \[
  [B]s_i = \text{true} \quad \text{and} \quad [M]s_i = s_{i+1} \quad \text{for each } i
  \]
So far we’ve looked at closed commands that don’t call any procedures. Let’s suppose there’s a parameterless procedure \( c \). Here’s the grammar of open commands, that are allowed to mention \( c \).

\[
N ::= M \mid N; N \mid \text{if } B \text{ then } N \text{ else } N \mid \text{while } B \text{ do } N \mid c()
\]
Procedure call

So far we’ve looked at closed commands that don’t call any procedures. Let’s suppose there’s a parameterless procedure \( c \). Here’s the grammar of open commands, that are allowed to mention \( c \).

\[
N ::= M | N; N | \text{if } B \text{ then } N \text{ else } N | \text{while } B \text{ do } N | c()
\]

Recall A closed command denotes an element of \( S \rightarrow S\perp \).
Procedure call

So far we’ve looked at closed commands that don’t call any procedures. Let’s suppose there’s a parameterless procedure c. Here’s the grammar of open commands, that are allowed to mention c.

\[
N ::= M \mid N; N \mid \text{if } B \text{ then } N \text{ else } N \mid \text{while } B \text{ do } N \mid c()\]

Recall A closed command denotes an element of \(S \rightarrow S_{\perp}\)

Suggestion An open command such as

\[
x := 3;\]
\[
\text{if } (y > 4) \text{ then } \{c()\} \text{ else } \{y := 2\}\]

denotes a function from \(S \rightarrow S_{\perp}\) to \(S \rightarrow S_{\perp}\). The argument to this function represents the meaning of c.
Procedure call

So far we’ve looked at closed commands that don’t call any procedures. Let’s suppose there’s a parameterless procedure \( c \). Here’s the grammar of open commands, that are allowed to mention \( c \).

\[
N ::= M \mid N; N \mid \text{if } B \text{ then } N \text{ else } N \mid \text{while } B \text{ do } N \mid c()
\]

Recall A closed command denotes an element of \( S \rightarrow S_\bot \)

Suggestion An open command such as

\[
\begin{align*}
x &:= 3; \\
\text{if } (y > 4) \text{ then } \{c()\} \text{ else } \{y := 2\}
\end{align*}
\]

denotes a function from \( S \rightarrow S_\bot \) to \( S \rightarrow S_\bot \).

The argument to this function represents the meaning of \( c \).

In fact, an open command must denote a continuous function. (Technical condition)
Recursive definition

Let’s extend the grammar of closed commands, so that we can define a closed command recursively.

\[
M ::= \text{skip} \mid x := E \mid y := E \mid \\
M ; M \mid \text{if } B \text{ then } M \text{ else } M \mid \text{while } B \text{ do } M \\
\mid \text{command } c() \{ N \}
\]

For example, here is a closed command:

\[
x := x + 5; \\
c\text{ommand } c() \{ \\
\quad x := 3; \\
\quad \text{if } (y > 4) \text{ then } \{ c() \} \text{ else } \{ y := 2 \} \\
\}; \\
y := 9
\]
Semantics of recursion

How can we give \([\text{command } c() \{N\}]\) in terms of \([N]\)?

For example, we want the closed command

\[
\begin{align*}
\text{command } c() & \{ \\
& \{x := 3; \\
& \text{if } (y > 4) \{ \\
& \quad c() \\
& \text{else } \\
& \quad y := 2 \}
\}
\end{align*}
\]

\] to denote an element of \(S \rightarrow S \perp\), mapping a state \((x, y)\) to \(\perp\) if \(y > 4\) the state \((3, 2)\) if \(y \leq 4\). How can we obtain this element from the denotation of the body?
Semantics of recursion

How can we give $[[\text{command } c() \{ N \}]]$ in terms of $[[N]]$?

For example, we want the closed command

\[
\text{command } c() \{
    x := 3;
    \text{if } (y > 4) \text{ then } \{ c() \} \text{ else } \{ y := 2 \}
\}
\]

to denote an element of $S \rightarrow S_{\bot}$, mapping a state $(x, y)$ to

- $\bot$ if $y > 4$
- the state $(3, 2)$ if $y \leq 4$.

How can we obtain this element from the denotation of the body?
A function $f$ from a set $A$ to itself is called an endofunction.

Is there an element $x \in A$ such that $f(x) = x$?

Such an element is called a fixpoint of $f$. 
A function $f$ from a set $A$ to itself is called an endofunction.

Is there an element $x \in A$ such that $f(x) = x$?

Such an element is called a fixpoint of $f$.

Examples of endofunctions on $\mathbb{Z}$

- $x \mapsto x + 1$ has no fixpoints.
- $x \mapsto 2x$ has one fixpoint.
- $x \mapsto x^2$ has two fixpoints.
- $x \mapsto x^3$ has three fixpoints.
- $x \mapsto x$ has infinitely many fixpoints.
Fixpoints and Recursion

An open command $N$ denotes a continuous endofunction on $S \rightarrow S_\bot$. The closed command `command c() {N}` must denote a fixpoint of that endofunction.
Fixpoints and Recursion

An open command $N$ denotes a continuous endofunction on $S \rightarrow S_\bot$. The closed command `command c() {N}` must denote a fixpoint of that endofunction.

But which fixpoint? For example the open command

\[
\begin{align*}
x & := 3; \\
\text{if } (y > 4) \text{ then } \{c()\} \text{ else } \{y := 2\}
\end{align*}
\]

denotes an endofunction with many fixpoints.

Here is a wrong fixpoint: the function that maps a state $(x,y)$ to the state $(y+2, y+7)$ if $y > 4$ the state $(3, 2)$ if $y \leq 4$.

The correct answer is the least fixpoint, i.e. as many $\bot$ s as possible.

It turns out that every continuous function has a least fixpoint. To model open commands as endofunctions, we need a suitable fixpoint theory.
An open command $N$ denotes a continuous endofunction on $S \rightarrow S_\bot$. The closed command $\text{command } c() \{N\}$ must denote a fixpoint of that endofunction.

But which fixpoint? For example the open command

\[
x := 3; \\
\text{if } (y > 4) \text{ then } \{c()\} \text{ else } \{y := 2\}
\]

denotes an endofunction with many fixpoints.

Here is a wrong fixpoint: the function that maps a state $(x, y)$ to

- the state $(y + 2, y + 7)$ if $y > 4$
- the state $(3, 2)$ if $y \leq 4$.

It turns out that every continuous function has a least fixpoint. To model open commands as endofunctions, we need a suitable fixpoint theory.
Fixpoints and Recursion

An open command $N$ denotes a continuous endofunction on $S \rightarrow S_{\bot}$. The closed command `command c() {N}` must denote a fixpoint of that endofunction.

But which fixpoint? For example the open command

\[
x := 3; \\
\text{if } (y > 4) \text{ then } \{c()\} \text{ else } \{y := 2\}
\]

denotes an endofunction with many fixpoints.

Here is a wrong fixpoint: the function that maps a state $(x, y)$ to

- the state $(y + 2, y + 7)$ if $y > 4$
- the state $(3, 2)$ if $y \leq 4$.

The correct answer is the least fixpoint, i.e. as many $\bot$s as possible.
Fixpoints and Recursion

An open command $N$ denotes a continuous endofunction on $S \rightarrow S_\bot$. The closed command `command c() {N}` must denote a fixpoint of that endofunction.

But which fixpoint? For example the open command

$$
x := 3;
\text{if } (y > 4) \text{ then } \{c()\} \text{ else } \{y := 2\}
$$

denotes an endofunction with many fixpoints.

Here is a wrong fixpoint: the function that maps a state $(x, y)$ to

- the state $(y + 2, y + 7)$ if $y > 4$
- the state $(3, 2)$ if $y \leq 4$.

The correct answer is the least fixpoint, i.e. as many $\bot$s as possible.

It turns out that every continuous function has a least fixpoint.
Fixpoints and Recursion

An open command $N$ denotes a continuous endofunction on $S \rightarrow S_{\bot}$. The closed command `command c() \{N\}` must denote a fixpoint of that endofunction.

But which fixpoint? For example the open command

```plaintext
x ::= 3;
if (y > 4) then \{c()\} else \{y ::= 2\}
```

denotes an endofunction with many fixpoints.

Here is a wrong fixpoint: the function that maps a state $(x, y)$ to
- the state $(y + 2, y + 7)$ if $y > 4$
- the state $(3, 2)$ if $y \leq 4$.

The correct answer is the least fixpoint, i.e. as many $\bot$s as possible.

It turns out that every continuous function has a least fixpoint.

To model open commands as endofunctions, we need a suitable fixpoint theory.
Equivalence

Note that we started off knowing what the meaning of a program should be, because we knew how to run a program.
Note that we started off knowing what the meaning of a program should be, because we knew how to run a program.

We also knew when we wanted two programs to be equivalent, i.e. to have the same meaning.
Equivalence

Note that we started off knowing what the meaning of a program should be, because we knew how to run a program.

We also knew when we wanted two programs to be equivalent, i.e. to have the same meaning.

An important question to address before formulating a denotational semantics is: when should two pieces of code be considered equivalent?
Equivalence

Note that we started off knowing what the meaning of a program should be, because we knew how to run a program.

We also knew when we wanted two programs to be equivalent, i.e. to have the same meaning.

An important question to address before formulating a denotational semantics is: when should two pieces of code be considered equivalent?

Ideally two pieces of code should have the same denotation if and only if they are equivalent in some \textit{a priori} sense.
Let’s say we add a printing commands to our language. Then a command, in a given starting state $s$, has three possible behaviours:

- to print a finite string $m$, then terminate in a state $s'$
- to print a finite string $m$, then diverge
- to print an infinite string $m$.

Let’s write $\text{Beh}$ for the set of behaviours.
Let’s say we add a printing commands to our language. Then a command, in a given starting state $s$, has three possible behaviours:

- to print a finite string $m$, then terminate in a state $s'$
- to print a finite string $m$, then diverge
- to print an infinite string $m$.

Let’s write Beh for the set of behaviours.

A closed command denotes an element of $S \rightarrow \text{Beh}$. An open command denotes a (continuous) endofunction on $S \rightarrow \text{Beh}$. 
We can consider many different programming language features, and try to come up with denotational models for them:

- higher-order functions (functions that take functions as parameters)
- data types
- recursively defined types
- input
- exceptions
- control operators
- local variables
- function variables
- different parameter-passing mechanisms.
Why consider nondeterminism?

So far, we’ve looked at programs that are deterministic: given the starting state, the behaviour follows inexorably.
Why consider nondeterminism?

So far, we’ve looked at programs that are deterministic: given the starting state, the behaviour follows inexorably.

But lots of programs in reality depend on hidden factors. Run them twice and they’ll do something different, for no intelligible reason.
Why consider nondeterminism?

So far, we’ve looked at programs that are deterministic: given the starting state, the behaviour follows inexorably.

But lots of programs in reality depend on hidden factors. Run them twice and they’ll do something different, for no intelligible reason.

- Perhaps because they involve concurrent threads, and the behaviour depends on the details of the scheduler, or on what other programs are being run by other users.
- Perhaps because they allocate some free memory, and the behaviour depends on which location is chosen.
- ...
Why consider nondeterminism?

So far, we’ve looked at programs that are deterministic: given the starting state, the behaviour follows inexorably.

But lots of programs in reality depend on hidden factors. Run them twice and they’ll do something different, for no intelligible reason.

- Perhaps because they involve concurrent threads, and the behaviour depends on the details of the scheduler, or on what other programs are being run by other users.
- Perhaps because they allocate some free memory, and the behaviour depends on which location is chosen.
- ...

The programmer has to assume that a program has a range of possible behaviours, and to ensure that all of them are acceptable.
Some nondeterministic constructs

There are various nondeterministic constructs we can put into a language. An example is or which chooses to go left or right:

\{x := 3; y := 4\} or \{x := 7\}

A more powerful construct is somenumber, which offers infinitely many possibilities:

\[ x := \text{somenumber}; \]
\[ \text{print "hello" x times} \]

Here is an attempt to achieve \(x := \text{somenumber}\) using just or.

\[
\begin{align*}
\text{local } z &:= 0 \\
z &:= 0 \text{ or } z := 1; \\
x &:= 0; \\
\text{while } (z = 0) \text{ do } \{ \\
& x := x + 1; \\
& \{z := 0\} \text{ or } \{z := 1\} \\
& \}
\end{align*}
\]

This may diverge.
Some nondeterministic constructs

There are various nondeterministic constructs we can put into a language. An example is \texttt{or} which chooses to go left or right:

\begin{verbatim}
{x := 3; y := 4} or {x := 7}
\end{verbatim}

A more powerful construct is \texttt{somenumber}, which offers infinitely many possibilities:

\begin{verbatim}
x := somenumber;
print "hello" x times
\end{verbatim}

Here is an attempt to achieve \texttt{x := somenumber} using just \texttt{or}.

\begin{verbatim}
local z := 0
z := 0 or z := 1;
x := 0;
while (z = 0) do {
x := x + 1;
{z := 0} or {z := 1}
}
//This may diverge.
\end{verbatim}
Suppose $E$ and $E'$ are two expressions that might return an integer or might diverge.
Suppose $E$ and $E'$ are two expressions that might return an integer or might diverge.

$E$ or $E'$ chooses to go left or right, and evaluates $E$ or $E'$ accordingly.

$E \text{ amb } E'$ evaluates $E$ and $E'$ concurrently, and returns whatever it gets first. This will diverge only if both $E$ and $E'$ diverge.
Suppose $E$ and $E'$ are two expressions that might return an integer or might diverge.

$E$ or $E'$ chooses to go left or right, and evaluates $E$ or $E'$ accordingly.

$E$ amb $E'$ evaluates $E$ and $E'$ concurrently, and returns whatever it gets first. This will diverge only if both $E$ and $E'$ diverge.

$$(3 \text{ or } 4) \text{ or } (3 \text{ or } 8 \text{ or } 9 \text{ or diverge})$$

can return 3, 4, 8 or 9 or diverge.

$$(3 \text{ or } 4) \text{ amb } (3 \text{ or } 8 \text{ or } 9 \text{ or diverge})$$

can return 3, 4, 8 or 9. It cannot diverge.

Amb is more powerful than some number.
Two programs are equivalent when they have the same properties. What properties should we consider?
Two programs are equivalent when they have the same **properties**. What properties should we consider?

The program must not kill the customer.
Two programs are equivalent when they have the same properties. What properties should we consider?

The program must not kill the customer.  
**safety property**
Two programs are equivalent when they have the same properties. What properties should we consider?

The program must not kill the customer.  
**safety property**

The program must greet the customer.
Two programs are equivalent when they have the same properties. What properties should we consider?

The program must not kill the customer.  
**safety property**

The program must greet the customer.  
**liveness property**
Two programs are equivalent when they have the same properties. What properties should we consider?

The program must not kill the customer.  
**safety property**

The program must greet the customer. 
**liveness property**

If the program insults the customer, it must apologize.
Two programs are equivalent when they have the same properties. What properties should we consider?

The program must not kill the customer.

safety property

The program must greet the customer.

liveness property

If the program insults the customer, it must apologize.

conditional liveness property
Two programs are equivalent when they have the same properties. What properties should we consider?

The program must not kill the customer.  
**safety property**

The program must greet the customer.  
**liveness property**

If the program insults the customer, it must apologize.  
**conditional liveness property**

The program must stop insulting the customer.
Two programs are equivalent when they have the same properties. What properties should we consider?

The program must not kill the customer.

safety property

The program must greet the customer.

liveness property

If the program insults the customer, it must apologize.

conditional liveness property

The program must stop insulting the customer.

infinite liveness property
Infinite trace equivalence

Let’s take a language with printing and nondeterminism. Two programs are infinite trace equivalent when they have the same range of behaviours for any starting state.
Let’s take a language with printing and nondeterminism. Two programs are infinite trace equivalent when they have the same range of behaviours for any starting state.

Probably the most obvious equivalence to consider.
Infinite trace equivalence

Let’s take a language with printing and nondeterminism. Two programs are infinite trace equivalent when they have the same range of behaviours for any starting state.

Probably the most obvious equivalence to consider.

Can recognize all the properties of our customer service program.
Infinite trace equivalence

Let’s take a language with printing and nondeterminism. Two programs are infinite trace equivalent when they have the same range of behaviours for any starting state.

Probably the most obvious equivalence to consider.

Can recognize all the properties of our customer service program.

Can we give a denotational semantics for this equivalence?
Let’s take a language with printing and nondeterminism. Two programs are infinite trace equivalent when they have the same range of behaviours for any starting state.

Probably the most obvious equivalence to consider.

Can recognize all the properties of our customer service program.

Can we give a denotational semantics for this equivalence?

A closed command will denote a relation from $S$ to Beh, i.e. an element of $S \rightarrow \mathcal{P}(\text{Beh})$. 
Let’s take a language with printing and nondeterminism. Two programs are infinite trace equivalent when they have the same range of behaviours for any starting state.

Probably the most obvious equivalence to consider.

Can recognize all the properties of our customer service program.

Can we give a denotational semantics for this equivalence?

A closed command will denote a relation from $S$ to Beh, i.e. an element of $S \rightarrow \mathcal{P}(\text{Beh})$.

What about an open command?
Could an open command denote an endofunction on $S \rightarrow \mathcal{P}$(Beh)?
Could an open command denote an endofunction on $S \rightarrow \mathcal{P}($Beh$)$? No
Could an open command denote an endofunction on $S \to \mathcal{P}(\text{Beh})$? No

Let’s say there’s just one character, ✓. Here’s an open command $N$

$\{$
\begin{align*}
    x &:= \text{somenumber}; \\
    \text{print } x \text{ ticks; } \\
    x &:= \text{somenumber}; \\
    y &:= \text{somenumber}; \\
    \{\text{skip or diverge}\} \\
\} \text{ or } \{c()\}$
Could an open command denote an endofunction on $S \rightarrow \mathcal{P}(\text{Beh})$? No

Let’s say there’s just one character, ✓. Here’s an open command $N$ and another one $N'$

```
{  
  x := somenumber;
  print x ticks;
  x := somenumber;
  y := somenumber;
  {skip or diverge}
} or {c()} or {print✓; c()}
```
In a given starting state $s$, can it print $n$ ticks and terminate in state $s'$? Always can it print $n$ ticks and diverge? Always can it print infinitely many ticks? Iff $c$ can do, $N$ and $N'$ have the same range of behaviours in any starting state.

Paul Blain Levy (University of Birmingham) Semantics of nondeterminism November 17, 2009 22 / 28
In a given starting state $s$

- can it print $n$ ticks and terminate in state $s'$?
Same endofunction

{ 
  x := somenumber;
  print x ticks;
  x := somenumber;
  y := somenumber;
  {skip or diverge}
} or {c()} or {print✓. c()}

In a given starting state $s$

- can it print $n$ ticks and terminate in state $s'$? Always
In a given starting state \( s \),

- can it print \( n \) ticks and terminate in state \( s' \)? Always \textbf{Always}

\[
\{ \\
    x := \text{somenumber}; \\
    \text{print \( x \) ticks}; \\
    x := \text{somenumber}; \\
    y := \text{somenumber}; \\
    \{ \text{skip or diverge} \} \\
\} \text{ or } \{ \text{c()} \} \text{ or } \{ \text{print✓. c()} \}
\]
Same endofunction

\[
\{ \\
    x := \text{somenumber}; \\
    \text{print } x \text{ ticks}; \\
    x := \text{somenumber}; \\
    y := \text{somenumber}; \\
    \{ \text{skip or diverge} \} \\
\} \text{ or } \{ \text{c()} \} \text{ or } \{ \text{print✓. c()} \}
\]

In a given starting state \(s\)

- can it print \(n\) ticks and terminate in state \(s'\)? Always \textbf{Always}
- can it print \(n\) ticks and diverge?
Same endofunction

{ 
  x := somenumber;
  print x ticks;
  x := somenumber;
  y := somenumber;
  {skip or diverge}
} or {c()} or {print✓. c()}

In a given starting state $s$
- can it print $n$ ticks and terminate in state $s'$? Always Always
- can it print $n$ ticks and diverge? Always
Same endofunction

{  
    x := somenumber;
    print x ticks;
    x := somenumber;
    y := somenumber;
    {skip or diverge}
} or \{c()\} or \{\text{print} \check\ . c()\}

In a given starting state $s$

- can it print $n$ ticks and terminate in state $s'$? Always Always
- can it print $n$ ticks and diverge? Always Always
In a given starting state $s$

- can it print $n$ ticks and terminate in state $s'$? Always **Always**
- can it print $n$ ticks and diverge? Always **Always**
- can it print infinitely many ticks?
Same endofunction

\{ \\
  x := somenumber; \\
  print x ticks; \\
  x := somenumber; \\
  y := somenumber; \\
  \{skip or diverge\} \\
  \} or \{c()\} or \{print✓. c()\}

In a given starting state $s$

- can it print $n$ ticks and terminate in state $s'$? Always Always
- can it print $n$ ticks and diverge? Always Always
- can it print infinitely many ticks? Iff c can
Same endofunction

\[
\begin{align*}
\{ & \\
& \quad x := \text{somenumerator}; \\
& \quad \text{print} \ x \ \text{ticks}; \\
& \quad x := \text{somenumerator}; \\
& \quad y := \text{somenumerator}; \\
& \quad \{\text{skip or diverge}\} \\
\} \ \text{or} \ \{c()\} \ \text{or} \ \{\text{print}✓. \ c()\}
\end{align*}
\]

In a given starting state \(s\)

- can it print \(n\) ticks and terminate in state \(s'\)? Always \textbf{Always}
- can it print \(n\) ticks and diverge? Always \textbf{Always}
- can it print infinitely many ticks? Iff \(c\) can \textbf{Iff} \(c\) can.
Same endofunction

{ 
    x := somenumber;
    print x ticks;
    x := somenumber;
    y := somenumber;
    {skip or diverge}
} or \{c()\} or \{\text{print✓.} \ c()\}

In a given starting state $s$

- can it print $n$ ticks and terminate in state $s'$? Always Always
- can it print $n$ ticks and diverge? Always Always
- can it print infinitely many ticks? Iff c can Iff c can.

Whatever c can do, $N$ and $N'$ have the same range of behaviours in any starting state.
Different fixpoints

Let’s apply the recursion operator to $N$

```plaintext
command c() {
    {
        x := somenumber;
        print x ticks;
        x := somenumber;
        y := somenumber;
        {skip or diverge}
    } or {c()}
}
```

In starting state $(0, 0)$, can this print infinitely many ticks?
Different fixpoints

Let’s apply the recursion operator to $N$

```plaintext
command c() {
    {
        x := somenumber;
        print x ticks;
        x := somenumber;
        y := somenumber;
        {skip or diverge}
    } or {c()}
}
```

In starting state (0, 0), can this print infinitely many ticks? No
Let’s apply the recursion operator to $N$ and to $N’$

```
command c() {
    {
        x := somenumber;
        print x ticks;
        x := somenumber;
        y := somenumber;
        {skip or diverge}
    } or {c()} or {print✓; c()}
}
```

In starting state $(0, 0)$, can this print infinitely many ticks? No
Different fixpoints

Let’s apply the recursion operator to $N$ and to $N’$

```plaintext
command c() {
    {
        x := somenumber;
        print x ticks;
        x := somenumber;
        y := somenumber;
        {skip or diverge}
    } or {c()} or {print✓; c()}
}
```

In starting state $(0, 0)$, can this print infinitely many ticks? No Yes
Solution [2005]: use game semantics

\[
\begin{array}{l}
\{ \\
  x := \text{somenumber}; \\
  \text{print } x \text{ ticks;} \\
  x := \text{somenumber}; \\
  y := \text{somenumber}; \\
  \{\text{skip or diverge}\} \\
\} \text{ or } \{c()\} \text{ or } \{\text{print✓. } c()\}
\end{array}
\]

\text{N and } N' \text{ must have different denotations.}
Solution [2005]: use game semantics

\{ 
x := \text{somenum}; 
\text{print } x \text{ ticks}; 
x := \text{somenum}; 
y := \text{somenum}; 
\{ \text{skip or diverge} \}
\} \text{ or } \{ \text{c()} \} \text{ or } \{ \text{print✓.} \text{c()} \}

\( N \) and \( N' \) must have different denotations.

\( N' \) can tick, then call its argument \( c \). \( N \) cannot.
An open behaviour might look like this.

- Proponent prints 3 ticks, then calls $c$ in state $(7, 3)$.
- Opponent returns in state $(5, 9)$.
- Proponent prints 7 ticks, then calls $c$ in state $(8, 8)$.
- Opponent returns in state $(1, 0)$.
- Proponent prints infinitely many ticks.

An open command denotes a function from states to open behaviours.
An open behaviour might look like this.

- Proponent prints 3 ticks, then calls c in state \((7, 3)\).
- Opponent returns in state \((5, 9)\).
- Proponent prints 7 ticks, then calls c in state \((8, 8)\).
- Opponent returns in state \((1, 0)\).
- Proponent prints infinitely many ticks.

An open command denotes a function from states to open behaviours. This gives us enough information to model recursion properly.
Bisimulation

We can also consider **branching time properties** that ask: at what point during execution are choices made?

```
print "a"
if print "b" or print "c"
else if print "a" then print "b"
else if print "a" then print "c"
```
Bisimulation

We can also consider branching time properties that ask: at what point during execution are choices made?

Example The program can print “a” and then be in a position where it can print “b” and can also print “c”.

\[
\text{print } "a"; \{\text{print } "b" \text{ or print } "c"\} \\
\{\text{print } "a"; \text{ print } "b" \} \text{ or } \{\text{print } "a"; \text{ print } "c"\}
\]

Two programs with the same branching time properties are bisimilar.
We can also consider **branching time properties** that ask: at what point during execution are choices made?

**Example** The program can print “a” and then be in a position where it can print “b” and can also print “c”.

```
print "a"; {print "b" or print "c"}
{print "a"; print "b" } or {print "a"; print "c"}
```

Two programs with the same branching time properties are **bisimilar**. This means they have the same **branching tree**.
Bisimulation

We can also consider branching time properties that ask: at what point during execution are choices made?

**Example** The program can print “a” and then be in a position where it can print “b” and can also print “c”.

```plaintext
print "a"; \{print "b" or print "c"\}
\{print "a"; print "b" \} or \{print "a"; print "c"\}
```

Two programs with the same branching time properties are bisimilar.

This means they have the same branching tree.

Let’s write Trees for the set of branching trees.
We can also consider **branching time properties** that ask: at what point during execution are choices made?

**Example**  The program can print “a” and then be in a position where it can print “b” and can also print “c”.

```plaintext
print "a"; {print "b" or print "c"}
{print "a"; print "b" } or {print "a"; print "c"}
```

Two programs with the same branching time properties are **bisimilar**.

This means they have the same **branching tree**.

Let’s write Trees for the set of branching trees.

A closed command should denote a function from $S$ to Trees.
We can also consider **branching time properties** that ask: at what point during execution are choices made?

**Example** The program can print “a” and then be in a position where it can print “b” and can also print “c”.

```plaintext
print "a"; {print "b" or print "c"} 
{print "a"; print "b" } or {print "a"; print "c"} 
```

Two programs with the same branching time properties are **bisimilar**. This means they have the same **branching tree**.

Let’s write Trees for the set of branching trees.

A closed command should denote a function from $S$ to Trees.

What about an open command?
Could an open command denote an endofunction on $S \rightarrow \text{Trees}$?
Could an open command denote an endofunction on $S \rightarrow \text{Trees}$?

Apparently
Context Lemma For Bisimilarity

Could an open command denote an endofunction on $S \rightarrow \text{Trees}$?

Apparently

**Theorem (the context lemma)**

Suppose that $N$ and $N'$ are open commands that, in any starting state, are bisimilar whatever $c$ may do

i.e. they represent the same endofunction on $S \rightarrow \text{Trees}$.

Then command $c() \{N\}$ and command $c() \{N'\}$ are bisimilar

i.e. they represent the same fixpoint.
Could an open command denote an endofunction on $S \rightarrow$ Trees?

Apparently

**Theorem (the context lemma)**

Suppose that $N$ and $N'$ are open commands that, in any starting state, are bisimilar whatever $c$ may do

i.e. they represent the same endofunction on $S \rightarrow$ Trees.

Then \texttt{command} $c() \{N\}$ and \texttt{command} $c() \{N'\}$ are bisimilar

i.e. they represent the same fixpoint.

The proof is elegant but mysterious.

It doesn’t tell us how to find that fixpoint, given the endofunction.
Could an open command denote an endofunction on $S \rightarrow \text{Trees}$? Apparently

**Theorem (the context lemma)**

Suppose that $N$ and $N'$ are open commands that, in any starting state, are bisimilar whatever $c$ may do.

i.e. they represent the same endofunction on $S \rightarrow \text{Trees}$.

Then command $c() \{N\}$ and command $c() \{N'\}$ are bisimilar

i.e. they represent the same fixpoint.

The proof is elegant but mysterious.

It doesn’t tell us how to find that fixpoint, given the endofunction.

**Possible research direction** Use “least” fixpoint with several different orderings.
Find semantics of bisimilarity.
Roscoe’s “Seeing Beyond Divergence” model of conditional liveness combines least and greatest fixpoint for recursion.
[2007] The context lemma holds if we include amb for integer expressions
[2007] but not if we include amb for expressions that return functions.
Functional languages
Relate to other kinds of semantics