Semantics of nondeterminism

Paul Blain Levy

University of Birmingham

November 17, 2009

Outline

Operational vs Denotational Semantics

How do we describe the meaning of a programming language?

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.

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 $[\![M]\!]$ for its denotation.

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 $[\![M]\!]$ 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.

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 $[\![M]\!]$ 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 `while` language

Our language has two (nonnegative) integer variables `x` and `y`.

Example: simple `while` language

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

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

Example: simple `while` language

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

$$E ::= \ \mathtt{x} \mid \mathtt{y} \mid E + E \mid E * E \mid n \qquad (n \in \mathbb{N})$$

Boolean expressions are given by the grammar

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

Example: simple `while` language

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

$$E ::= \mathtt{x} \mid \mathtt{y} \mid E + E \mid E * E \mid n \qquad (n \in \mathbb{N})$$

Boolean expressions are given by the grammar

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

Commands are given by the grammar

$$\begin{array}{rcl} M & ::= & \mathtt{skip} \mid \mathtt{x}{:=}E \mid \mathtt{y}{:=}E \\ & & M; M \mid \mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ M \mid \mathtt{while}\ B\ \mathtt{do}\ M \end{array}$$

Semantics of Expressions

A state is a pair of integers e.g. $(4, 17)$.
This means that currently $\mathtt{x} = 4$ and $\mathtt{y} = 17$.

Semantics of Expressions

A state is a pair of integers e.g. $(4, 17)$.
This means that currently $\mathtt{x} = 4$ and $\mathtt{y} = 17$.

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

Semantics of Expressions

A state is a pair of integers e.g. $(4, 17)$.
This means that currently $\mathtt{x} = 4$ and $\mathtt{y} = 17$.

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

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

Semantics of Expressions

A state is a pair of integers e.g. $(4, 17)$.
This means that currently $\mathtt{x} = 4$ and $\mathtt{y} = 17$.

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

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 $[\![E]\!]s + [\![E']\!]s$.

Semantics of Expressions

A state is a pair of integers e.g. $(4, 17)$.
This means that currently $\mathtt{x} = 4$ and $\mathtt{y} = 17$.

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

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).

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).

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 $[\![M]\!]$ from S to $S_\perp$.

For example, we want the denotation of

$$\begin{array}{l} \mathtt{x} := \mathtt{x} + 4; \\ \mathtt{while}\ (\mathtt{x} > \mathtt{y})\ \mathtt{do}\ \{\mathtt{x} := \mathtt{x} + 1\} \end{array}$$

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

- $\perp$ if $x + 4 > y$
- $(x + 4, y)$ if $x + 4 \leqslant y$.

Example: the meaning of `while`

$[\![\texttt{while } B \texttt{ do } M]\!]$ 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

$$\begin{array}{rlll} [\![B]\!]s_i = \text{true} & \text{and} & [\![M]\!]s_i = s_{i+1} & \text{for each } i < n \\ [\![B]\!]s_n = \text{false} & & & \end{array}$$

- $\perp$ if there is a sequence of states $s = s_0, s_1, \ldots, s_n$ such that

$$\begin{array}{rlll} [\![B]\!]s_i = \text{true} & \text{and} & [\![M]\!]s_i = s_{i+1} & \text{for each } i < n \\ [\![B]\!]s_n = \text{true} & \text{and} & [\![M]\!]s_n = \perp & \end{array}$$

- $\perp$ if there is an infinite sequence of states $s = s_0, s_1, \ldots$ such that

$$[\![B]\!]s_i = \text{true} \quad \text{and} \quad [\![M]\!]s_i = s_{i+1} \qquad \text{for each } i$$

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.

$$N \quad ::= \quad M \mid N; N \mid \texttt{if } B \texttt{ then } N \texttt{ else } N \mid \texttt{while } B \texttt{ do } N \mid \mathrm{c}()$$

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.

$$N \quad ::= \quad M \mid N; N \mid \texttt{if } B \texttt{ then } N \texttt{ else } N \mid \texttt{while } B \texttt{ do } N \mid \mathtt{c}()$$

Recall A closed command denotes an element of $S \to S_\bot$

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.

$$N \quad ::= \quad M \mid N; N \mid \texttt{if } B \texttt{ then } N \texttt{ else } N \mid \texttt{while } B \texttt{ do } N \mid \mathtt{c}()$$

Recall A closed command denotes an element of $S \to S_\perp$
Suggestion An open command such as

$$\begin{array}{l} \mathrm{x} := 3; \\ \texttt{if } (\mathrm{y} > 4) \texttt{ then } \{\mathrm{c}()\} \texttt{ else } \{\mathrm{y} := 2\} \end{array}$$

denotes a function from $S \to S_\perp$ to $S \to S_\perp$.
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.

$$N \quad ::= \quad M \mid N; N \mid \texttt{if } B \texttt{ then } N \texttt{ else } N \mid \texttt{while } B \texttt{ do } N \mid \texttt{c()}$$

Recall A closed command denotes an element of $S \to S_\perp$

Suggestion An open command such as

$$\begin{array}{l} \mathtt{x} := 3; \\ \texttt{if } (\mathtt{y} > 4) \texttt{ then } \{\mathtt{c}()\} \texttt{ else } \{\mathtt{y} := 2\} \end{array}$$

denotes a function from $S \to S_\perp$ to $S \to S_\perp$.
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.

$$
\begin{array}{rcl}
M & ::= & \mathtt{skip} \mid \mathrm{x}{:=}E \mid \mathrm{y}{:=}E \mid \\
 & & M; M \mid \mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ M \mid \mathtt{while}\ B\ \mathtt{do}\ M \\
 & & \mid \mathtt{command}\ \mathtt{c}()\ \{N\}
\end{array}
$$

For example, here is a closed command:

$$
\begin{array}{l}
x := x + 5; \\
\mathtt{command}\ \mathtt{c}()\ \{ \\
\quad \mathrm{x} := 3; \\
\quad \mathtt{if}\ (\mathrm{y} > 4)\ \mathtt{then}\ \{\mathtt{c}()\}\ \mathtt{else}\ \{\mathrm{y} := 2\} \\
\}; \\
y := 9
\end{array}
$$

Semantics of recursion

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

Semantics of recursion

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

For example, we want the closed command

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

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

- $\perp$ if $y > 4$
- the state $(3, 2)$ if $y \leqslant 4$.

How can we obtain this element from the denotation of the body?

Fixpoints

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.

Fixpoints

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 $S \to S_\perp$.
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 $S \to S_\perp$. The closed command `command c()` $\{N\}$ must denote a fixpoint of that endofunction.

But which fixpoint? For example the open command

$$
\begin{array}{l}
\mathtt{x} := 3; \\
\mathtt{if}\ (\mathtt{y} > 4)\ \mathtt{then}\ \{\mathtt{c}()\}\ \mathtt{else}\ \{\mathtt{y} := 2\}
\end{array}
$$

denotes an endofunction with many fixpoints.

Fixpoints and Recursion

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

But which fixpoint? For example the open command

```
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 $y \leqslant 4$.

Fixpoints and Recursion

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

But which fixpoint? For example the open command

$$\begin{array}{l}\mathtt{x} := 3;\\ \mathtt{if}\ (\mathtt{y} > 4)\ \mathtt{then}\ \{\mathtt{c}()\}\ \mathtt{else}\ \{\mathtt{y} := 2\}\end{array}$$

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 $y \leqslant 4$.

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

Fixpoints and Recursion

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

But which fixpoint? For example the open command

$$\begin{array}{l} \mathtt{x} := 3; \\ \mathtt{if}\ (\mathtt{y} > 4)\ \mathtt{then}\ \{\mathtt{c()}\}\ \mathtt{else}\ \{\mathtt{y} := 2\} \end{array}$$

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 $y \leqslant 4$.

The correct answer is the least fixpoint, i.e. as many $\perp$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 $S \to S_\perp$. The closed command `command c()` $\{N\}$ must denote a fixpoint of that endofunction.

But which fixpoint? For example the open command

$$\begin{array}{l} \mathtt{x} := 3; \\ \mathtt{if}\ (\mathtt{y} > 4)\ \mathtt{then}\ \{\mathtt{c}()\}\ \mathtt{else}\ \{\mathtt{y} := 2\} \end{array}$$

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 $y \leqslant 4$.

The correct answer is the least fixpoint, i.e. as many $\perp$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.

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.

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 *a priori* sense.

Printing

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.

Printing

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 $S \rightarrow \mathrm{Beh}$.
An open command denotes a (continuous) endofunction on $S \rightarrow \mathrm{Beh}$.

Additional Features

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:

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

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

```
x := somenumber;
print "hello" x times
```

Here is an attempt to achieve `x := somenumber` using just `or`.

```
local z := 0
z := 0 or z := 1;
x := 0;
while (z = 0) do {
    x := x + 1;
    {z := 0} or {z := 1}
}
```

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:

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

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

```
x := somenumber;
print "hello" x times
```

Here is an attempt to achieve `x := somenumber` using just `or`.

```
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.
```

Erratic vs Ambiguous Nondeterminism

Suppose E and E' are two expressions that might return an integer or might diverge.

Erratic vs Ambiguous Nondeterminism

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.

Erratic vs Ambiguous Nondeterminism

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.

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

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

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

can return 3, 4, 8 or 9. It cannot diverge.
Amb is more powerful than `somenumber`.

Example Application

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

Example Application

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

The program must not kill the customer.

Example Application

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

The program must not kill the customer.
safety property

Example Application

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.

Example Application

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

Example Application

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.

Example Application

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

Example Application

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.

Example Application

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.

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.

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?

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?

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

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?

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

What about an open command?

Infinite traces and endofunctions

Could an open command denote an endofunction on $S \to \mathcal{P}(\mathsf{Beh})$?

Infinite traces and endofunctions

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

Infinite traces and endofunctions

Could an open command denote an endofunction on $S \to \mathcal{P}(\mathrm{Beh})$? No
Let's say there's just one character, ✓. Here's an open command N

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

Infinite traces and endofunctions

Could an open command denote an endofunction on $S \to \mathcal{P}(\mathrm{Beh})$? 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()}
```

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

Same endofunction

$$
\begin{array}{l}
\{ \\
\quad \mathtt{x} := \mathtt{somenumber}; \\
\quad \mathtt{print\ x\ ticks}; \\
\quad \mathtt{x} := \mathtt{somenumber}; \\
\quad \mathtt{y} := \mathtt{somenumber}; \\
\quad \{\mathtt{skip\ or\ diverge}\} \\
\}\ \mathtt{or}\ \{\mathtt{c}()\}\ \mathtt{or}\ \{\mathtt{print}\checkmark.\ \mathtt{c}()\}
\end{array}
$$

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

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

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?

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 {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

```
{
    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?

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

$$\begin{array}{l} \{ \\ \quad \texttt{x} := \texttt{somenumber}; \\ \quad \texttt{print x ticks}; \\ \quad \texttt{x} := \texttt{somenumber}; \\ \quad \texttt{y} := \texttt{somenumber}; \\ \quad \{\texttt{skip or diverge}\} \\ \}\ \texttt{or}\ \{\mathsf{c}()\}\ \texttt{or}\ \{\texttt{print}\checkmark.\ \mathsf{c}()\} \end{array}$$

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.

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 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

```
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

```
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

Different fixpoints

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'

```
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

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

N and N' must have different denotations.

Solution [2005]: use game semantics

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

N and N' must have different denotations.

N' can tick, then call its argument c. N cannot.

Game semantics (continued)

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.

Game semantics (continued)

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?

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".

$$\texttt{print "a"; \{print "b" or print "c"\}}$$

$$\texttt{\{print "a"; print "b" \} or \{print "a"; print "c"\}}$$

Two programs with the same branching time properties are bisimilar.

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".

$$\texttt{print}\ \text{"a"};\ \{\texttt{print}\ \text{"b"}\ \texttt{or}\ \texttt{print}\ \text{"c"}\}$$

$$\{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"b"}\ \}\ \texttt{or}\ \{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"c"}\}$$

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".

$$\texttt{print}\ \text{"a"};\ \{\texttt{print}\ \text{"b"}\ \texttt{or}\ \texttt{print}\ \text{"c"}\}$$

$$\{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"b"}\ \}\ \texttt{or}\ \{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"c"}\}$$

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.

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".

$$\texttt{print}\ \text{"a"};\ \{\texttt{print}\ \text{"b"}\ \texttt{or}\ \texttt{print}\ \text{"c"}\}$$

$$\{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"b"}\ \}\ \texttt{or}\ \{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"c"}\}$$

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.

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".

$$\texttt{print}\ \text{"a"};\ \{\texttt{print}\ \text{"b"}\ \texttt{or}\ \texttt{print}\ \text{"c"}\}$$
$$\{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"b"}\ \}\ \texttt{or}\ \{\texttt{print}\ \text{"a"};\ \texttt{print}\ \text{"c"}\}$$

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?

Context Lemma For Bisimilarity

Could an open command denote an endofunction on $S \to$ Trees?

Context Lemma For Bisimilarity

Could an open command denote an endofunction on $S \to$ Trees?
Apparently

Context Lemma For Bisimilarity

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 `command c() {N}` and `command c() {N'}` are bisimilar
i.e. they represent the same fixpoint.

Context Lemma For Bisimilarity

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 `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.

Context Lemma For Bisimilarity

Could an open command denote an endofunction on $S \to$ 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 $S \to$ 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.

New directions in nondeterminism

- 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