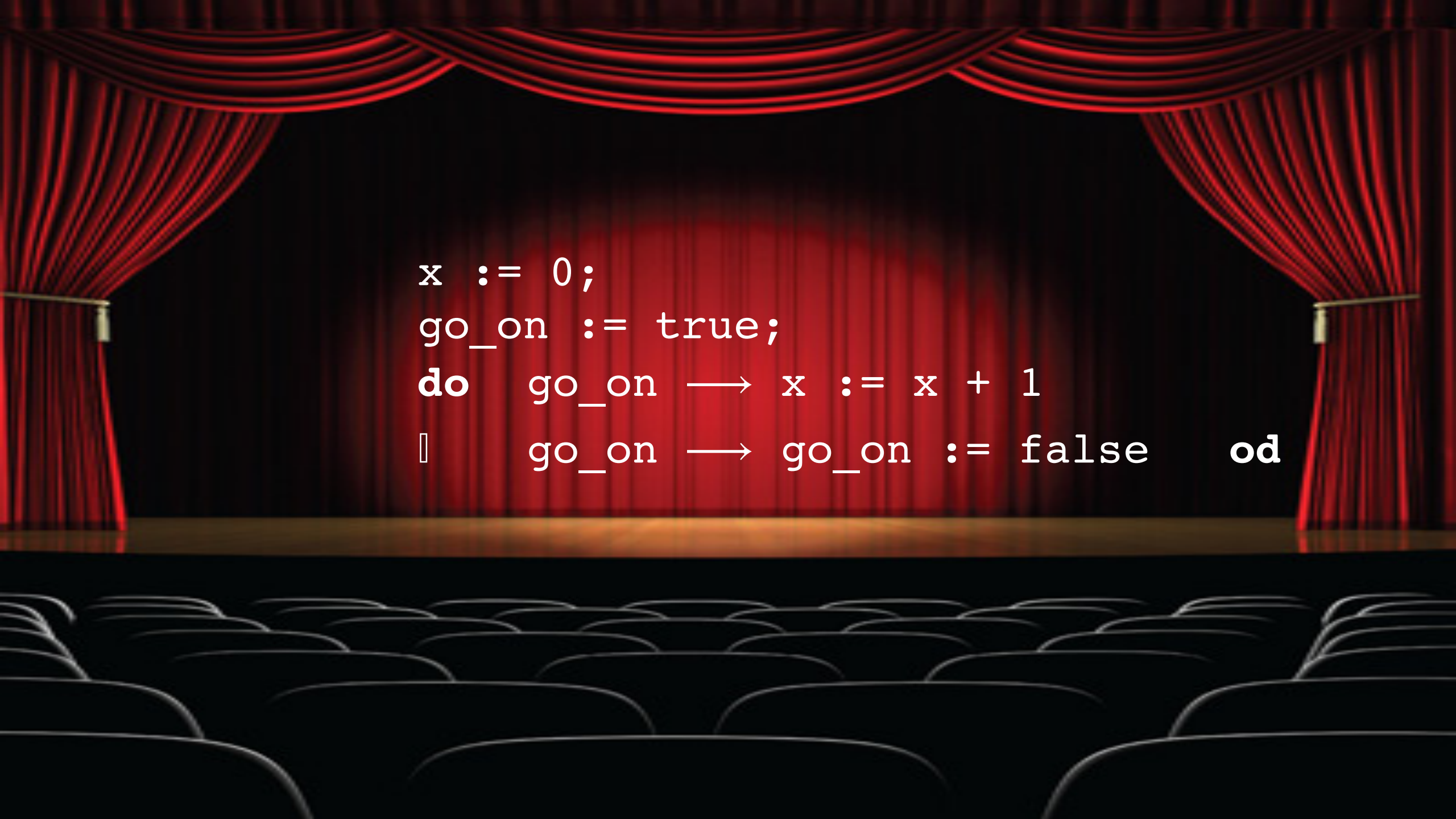
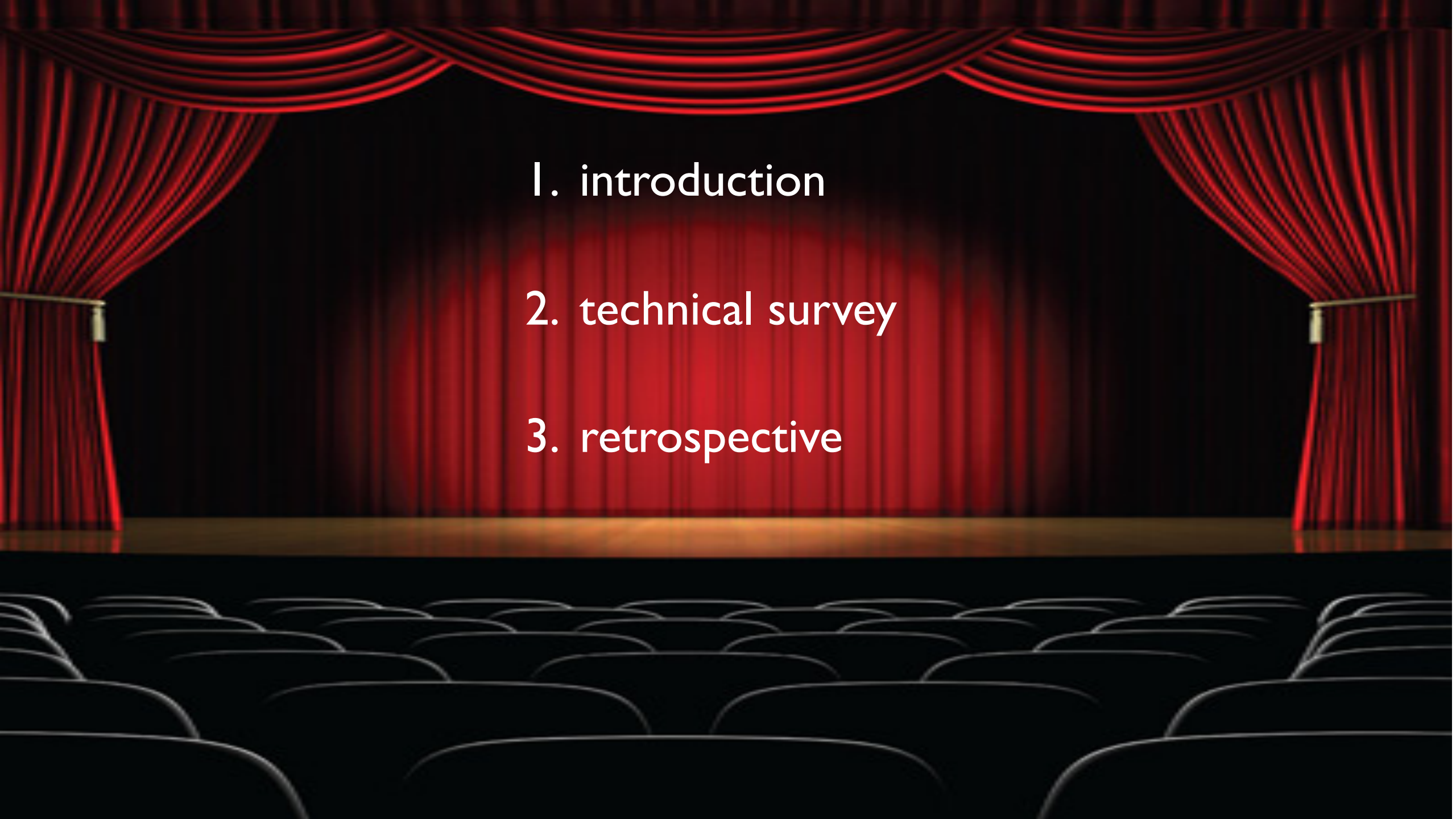
Unbounded Nondeterminism:

a Landscape for the Philosopher of Computing

Felice Cardone (Torino)
felice.cardone@unito.it
Edgar Daylight (Lille & Siegen)
egdaylight@dijkstrascry.com

PROGRAMme Spring Workshop (Lille)
6 June 2019


x := 0;
go_on := true;
do go_on ⟶ x := x + 1
▯ go_on ⟶ go_on := false od

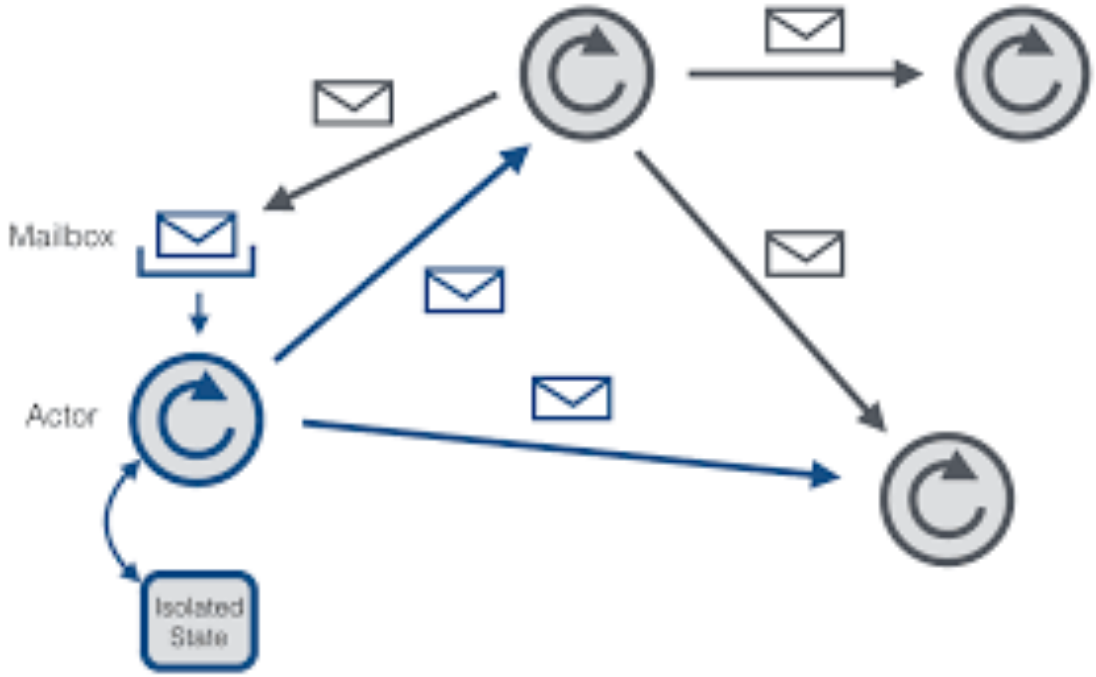

1. introduction
2. technical survey
3. retrospective

Computing is largely about …

physics

interrupts

asynchronous communication



Computing is largely about …

physics

interrupts

asynchronous communication

HEWITT

DIJKSTRA

LAMPORT

Carl Hewitt (2009)

"… asynchronous communication cannot be implemented by Turing machines because the order of arrival of messages cannot be logically inferred …"

Carl Hewitt (2009)

"Decisive paradigm shift when the notion of an **interrupt** was invented so that input that arrived asynchronously from outside could be incorporated in an ongoing computation"

Scrutinizing Hewitt's "paradigm shift"

pluralistically

technically

retrospectively

Scrutinizing Hewitt's "paradigm shift"

pluralistically :: In which ways did historical actors disagree?

technically :: What constitutes the mathematical landscape?

retrospectively :: How has our understanding increased?

Scrutinizing Hewitt's "paradigm shift"

pluralistically :: In which ways did historical actors disagree?

technically :: What constitutes the mathematical landscape?

retrospectively :: How has our understanding increased?

pluralistically :: actors disagreed

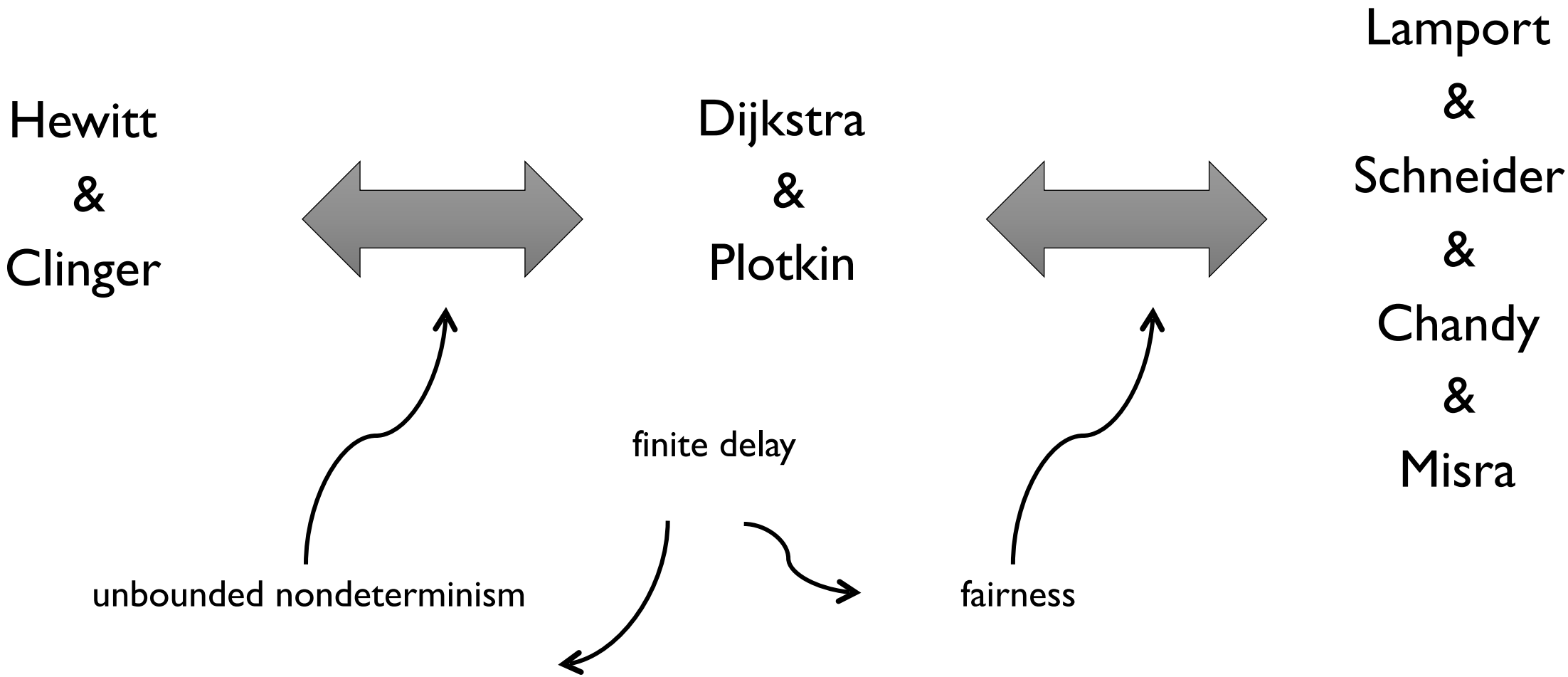
pluralistically :: actors disagreed

Hewitt
&
Clinger

Dijkstra
&
Plotkin

Lamport
&
Schneider
&
Chandy
&
Misra

pluralistically :: actors disagreed

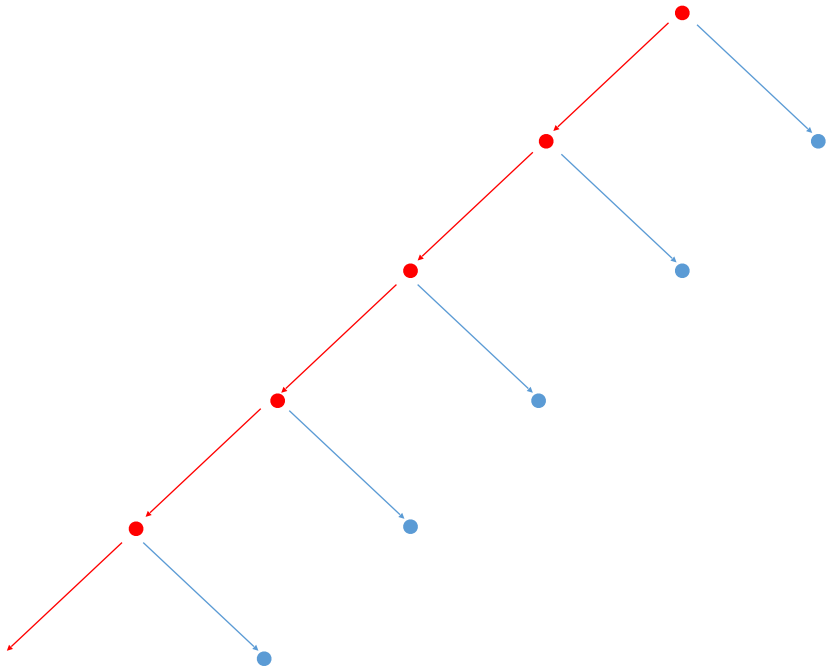


technically :: 3 core concepts

finite delay

unbounded nondeterminism

fairness



technically :: a mathematical landscape

Dijkstra :: *guarded commands*

McCarthy :: *ambiguous functions*

Keller :: *labeled transition systems*

Hoare :: *angelic vs. demonic nondeterminism*

Park :: *tight vs. loose nondeterminism*

retrospectively :: unbounded nondeterminism

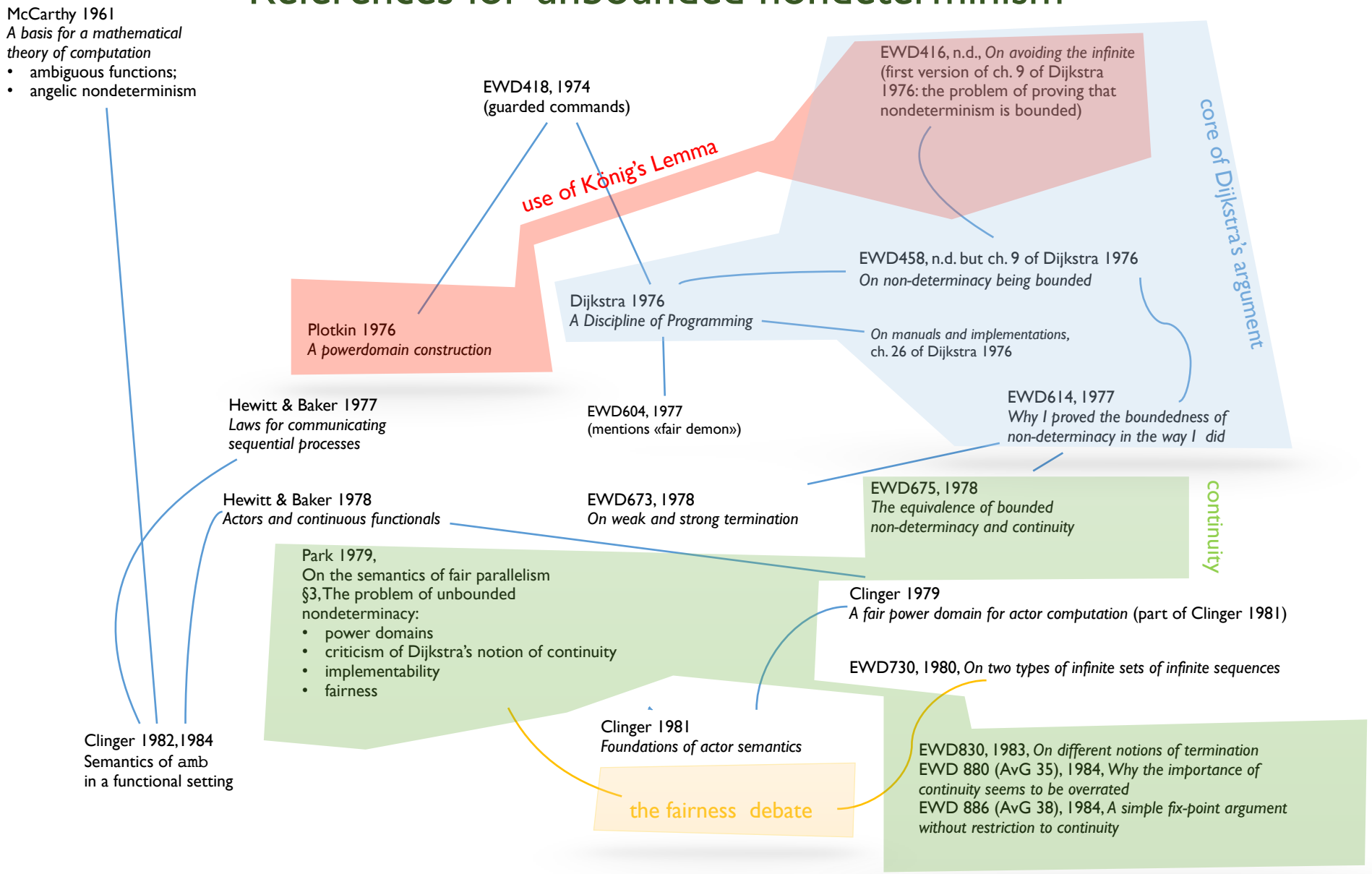
- Can we “implement” it?
- Is it stronger than Turing-machine nondeterminism?

1. introduction

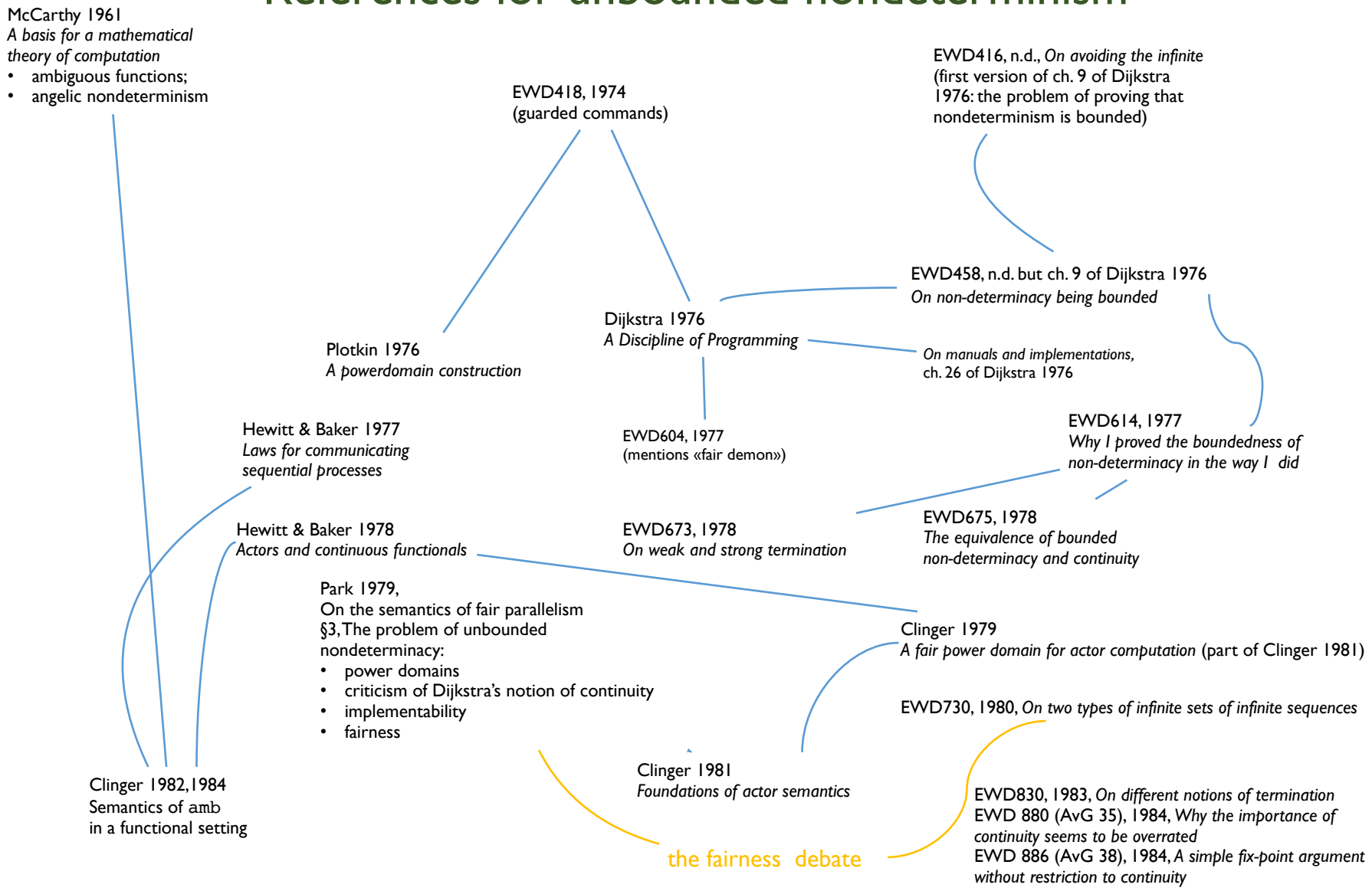
2. technical survey

3. retrospective

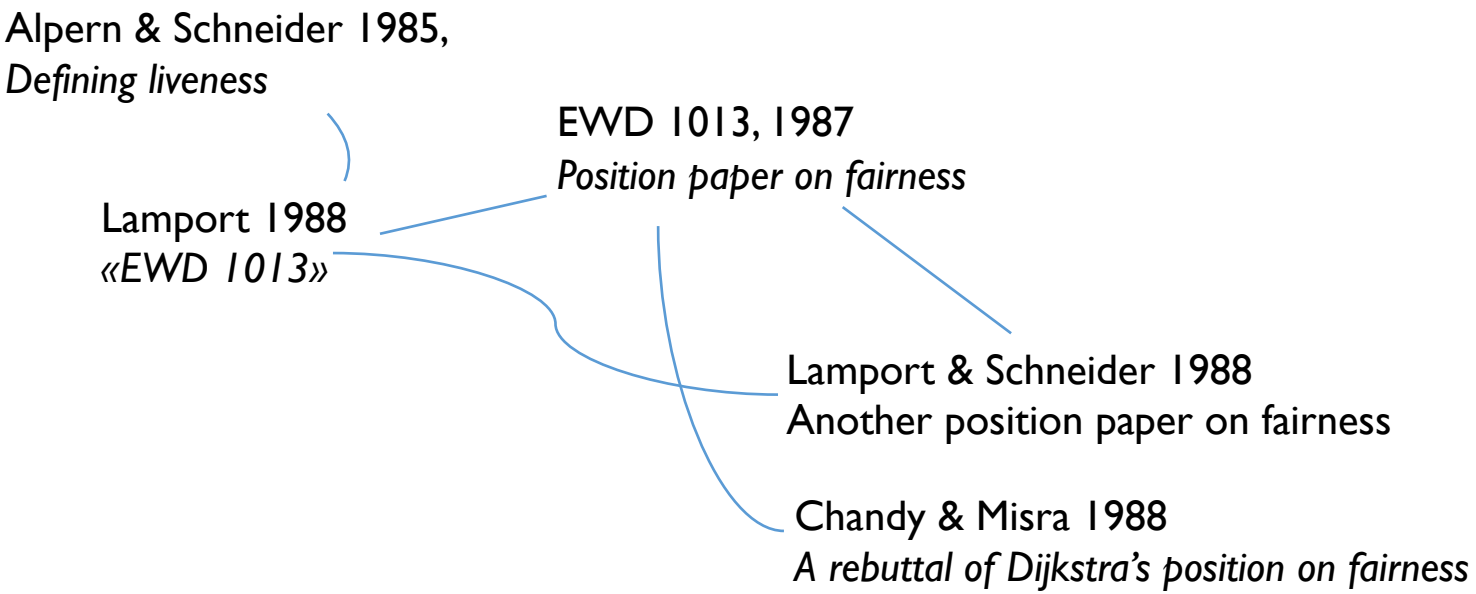
References for unbounded nondeterminism



References for unbounded nondeterminism



References for the fairness debate



Dijkstra's guarded commands

Repetitive construct

```
DO:    do
              G   ⟶  S
       ▯      G′  ⟶  S′
       od
```

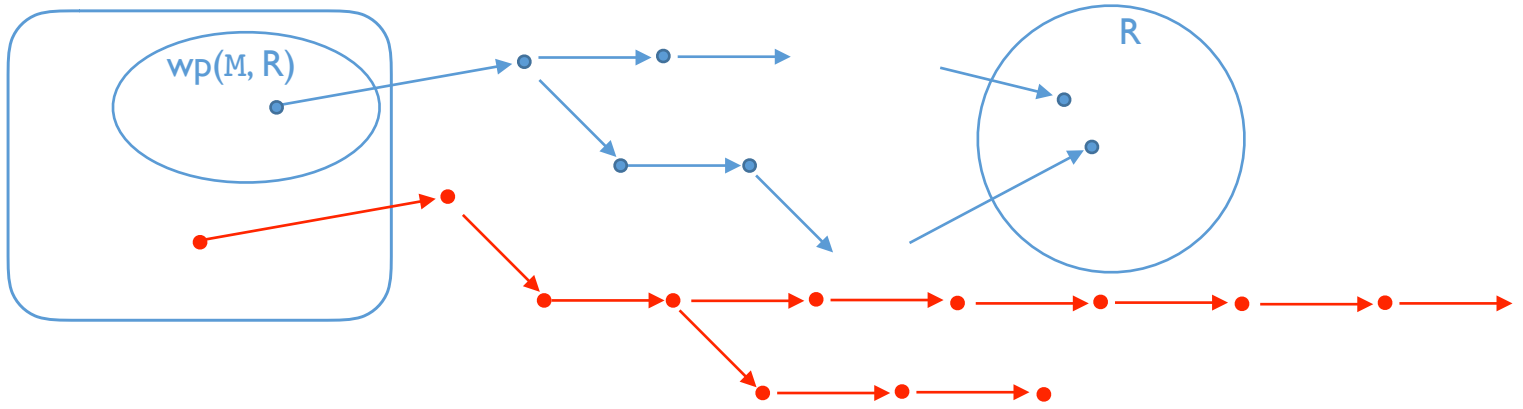
the boolean expressions ("guards") `G` and `G′` may be simultaneously true

☛ (finite) nondeterminism

Dijkstra's program semantics: definitions

Predicate transformers for a mechanism M

a function mapping each post-condition R describing a set of final states to its weakest pre-condition wp(M, R) satisfied by all and only the states such that the activation of M in each of these initial states will certainly result in a properly terminating happening leaving the system in a final state satisfying R.



Dijkstra's program semantics: definitions

Predicate transformer for repetitive construct

$$wp(\text{DO},R) = \exists k \geq 0\ H_k(R)$$

where

$H_k(R)$ holds of a state s, intuitively, if the construct terminates in k steps yielding a state satisfying R.

Dijkstra's program semantics: definitions

Continuity of a mechanism M

For any sequence of predicates $C_0, C_1, C_2, \ldots$ where $C_i \longrightarrow C_{i+1}$

$$wp(M, \exists k \geq 0\ C_k) = \exists k \geq 0\ wp(M, C_k)$$

The problem with unbounded nondeterminism

Random assignment

`x := ?` (? any natural number)

Example

```
S:    do
            x > 0 ⟶ x := x - 1
      ▯     x < 0 ⟶ x := ?
      od
```

Problem

- This program always terminates, with `x = 0`,
- there is no upper bound on the number of computation steps when `x < 0`,
- but in Dijkstra's semantics, termination is not guaranteed in this case.

The problem with unbounded nondeterminism

Dijkstra's mechanism:

```
S:    x := 0; go_on := true;
      do
            go_on ⟶ x := x + 1
      ▯     go_on ⟶ go_on := false
      od
```

For Dijkstra's mechanism S, the possible final states (if any) are:

$x = 0, x = 1, x = 2, \ldots$

Requiring necessary termination of S yields unbounded nondeterminism. The mechanism:

1. always terminates,
2. has denumerably many final states.

The problem with unbounded nondeterminism

Dijkstra's mechanism:

```
S:    x := 0; go_on := true;
      do
            go_on ⟶ x := x + 1
      ▯     go_on ⟶ go_on := false
      od
```

This mechanism:

- implements random assignment: `x := ?`
- operationally it is weakly terminating but not strongly terminating (Dijkstra EWD673,675): there is no upper bound on the number of steps leading from x = 0 to x = ?
- wp(S,x ≥ 0) = true but wp(S,x ≤ k) = false, for all natural values of k.

wp(S,) is not continuous

On implementability

❝ If programs are meant to be run on a hypothetical unbounded machine, we may still hope that, for every fixed program S in a given initial state, the integers manipulated by S belong to a bounded interval: in this case the existence of a physical machine implementing S is not a priori impossible. In order to meet this requirement, however, non-determinism must be bounded ❞

(EWD416)

A similar concern brought C.A. Petri in 1962 to the notion of concurrent computation.

❝ A mechanism of unbounded nondeterminacy yet guaranteed to terminate would be able to make within a finite time a choice out of infinitely many possibilities: if such a mechanism could be formulated in our programming language, that very fact would present an insurmountable barrier to the possibility of the implementation of that programming language ❞

(EWD614)

Dijkstra's strategy in EWD458

Dijkstra exploited the following strategy for proving the boundedness of nondeterminism:

- Define wp(M,) for all mechanisms M, by structural induction;
- Prove that wp(M,) is continuous for any mechanism M, namely:

  For any sequence of predicates $C_0, C_1, C_2, \ldots$ where $C_i \longrightarrow C_{i+1}$

  $$wp(M, \exists k \geq 0\ C_k) = \exists k \geq 0\ wp(M, C_k)$$

- Show that mechanism S is not continuous if unbounded nondeterminism is assumed;

$\therefore$ Nondeterminism is bounded

McCarthy's amb (1961)

Ambiguous "functions"

$amb(x,\perp) = x$ $\qquad$ $amb(v_1, v_2) = v_1$

$amb(\perp,y) = y$ $\qquad$ $amb(v_1, v_2) = v_2$

where $\perp$ interprets divergence.

Examples

Finite nondeterministic choice (Floyd 1967)

$choice(N) =_{def}$ **if** $N = 1$ **then** 1 **else** $amb(N, choice(N - 1))$

Denumerable nondeterministic choice

Let $f(n) = amb(n,f(n + 1))$

then the evaluation of f(n) always terminates, and for $? =_{def} f(0)$ we have

$? = 0$ or $? = 1$ or $? = 2$ or …

A formal setting

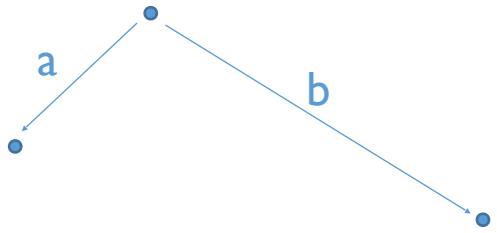
(Labeled) transition systems (Keller)

There is a set Q of states, a set A of transition labels and (binary) transition relations $\longrightarrow_a$ (one for each a in A).

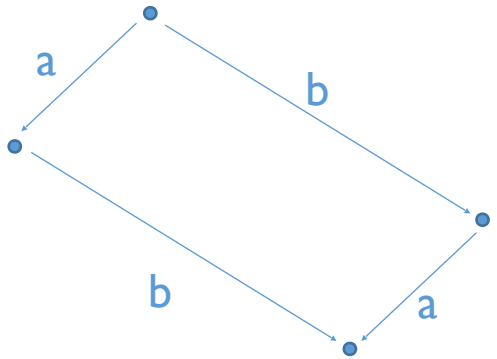
$q \longrightarrow_a q'$ means: the atomic action a can lead from state q to state q'.

If there is q' related this way to q, say that a is enabled in q and that q is active.

nondeterminism



concurrency (as interleaving)



The finite delay property

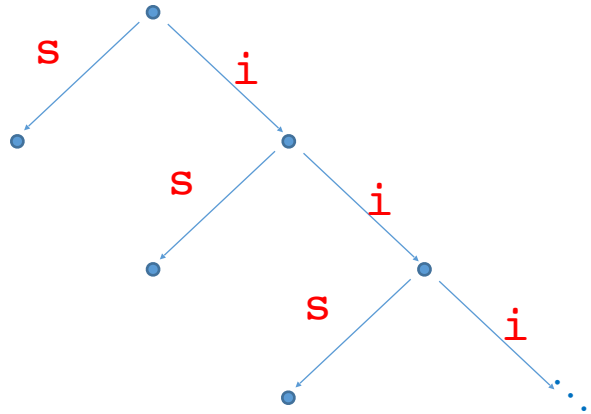
Formulation taken from Keller of a property introduced in the early studies of parallel computation by Karp, Miller and Keller:

> “ in a transition system no transition can be forever enabled without occurring ”

Finite delay property $\Longrightarrow$ Fairness (and therefore also unbounded nondeterminism)

For Dijkstra’s mechanism

```
S:      x := 0; go_on := true;
        do
                        i : go_on ⟶ x := x + 1
        ▯               s : go_on ⟶ go_on := false
        od
```

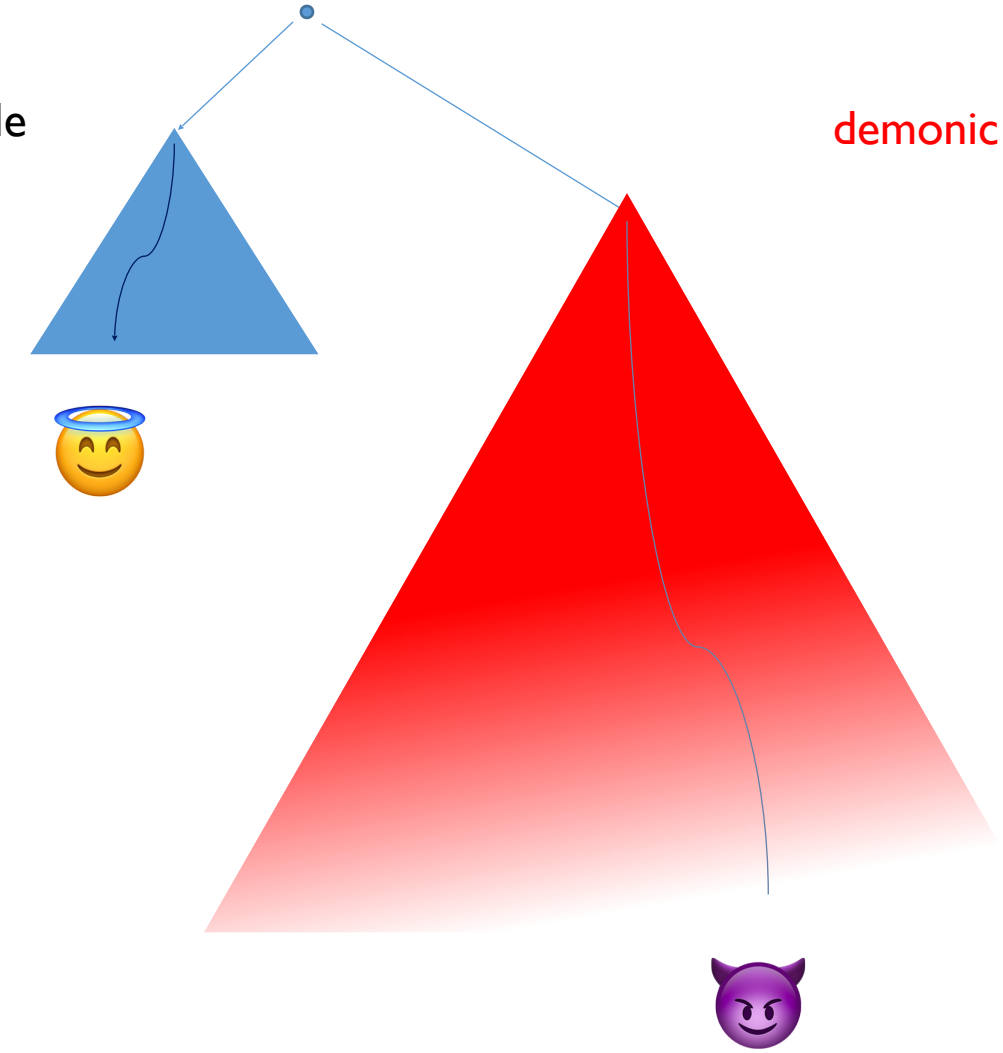


transition s is enabled at any state along the spine, and must occur eventually.

Two axes of variation:
Angelic vs demonic nondeterminism (Hoare)

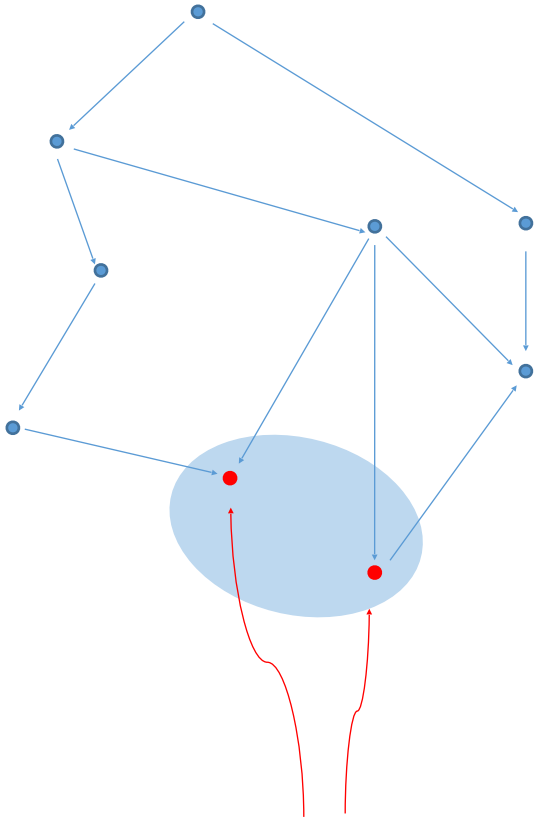
angelic = terminates if possible

demonic = diverges if possible



Two axes of variation: Tight vs loose nondeterminism (Park)

semantical specification

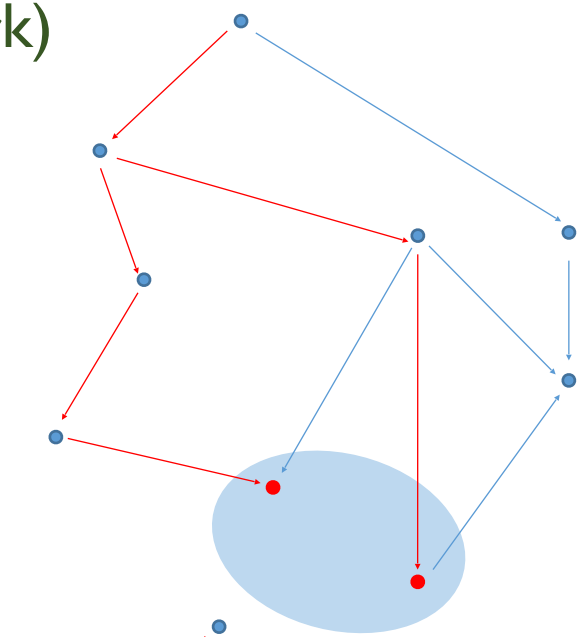


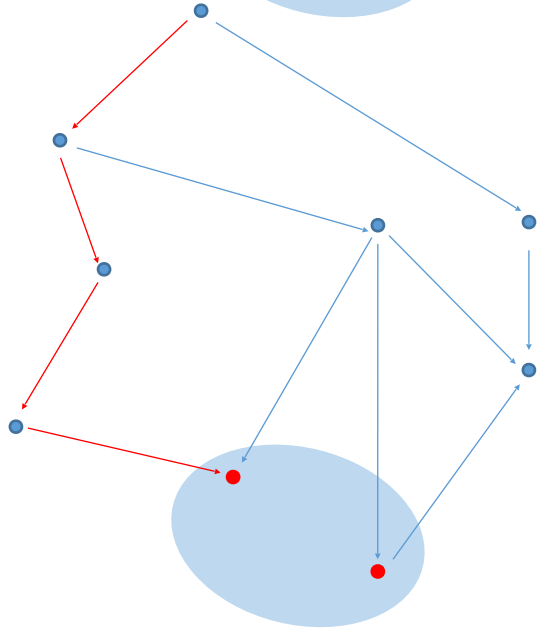
possible results

tight nondeterminism:
all possible results according to the semantics are reachable

loose nondeterminism:
all reachable results are admitted by the semantics

implementations





Operational argument against unbounded nondeterminism

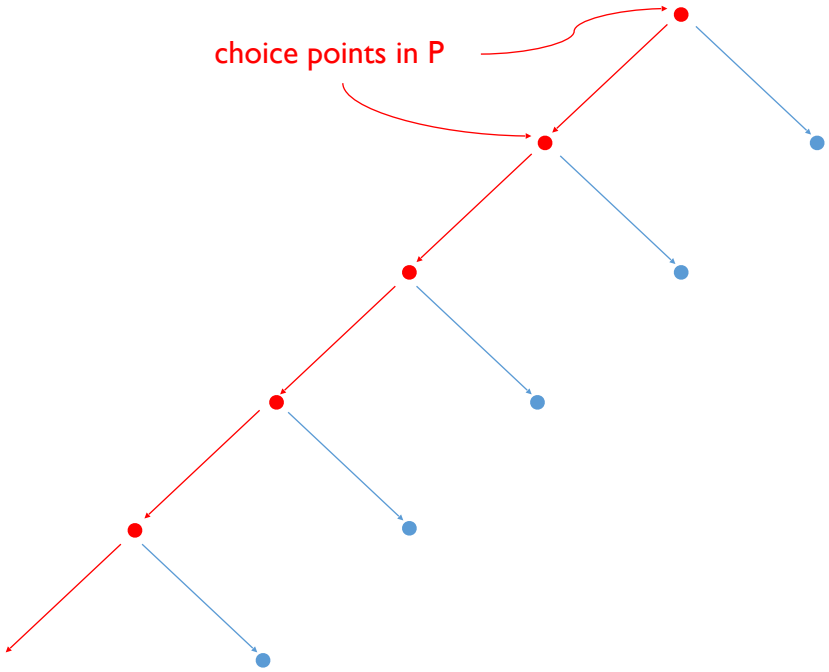
Dijkstra; Plotkin 1976:

❝ Now the set of all initial segments of execution sequences of a given nondeterministic program P, starting from a given state, will form a tree

The tree is finitary

Now König's lemma says that if every branch of a finitary tree is finite, then so is the tree itself

∴ So if an output set of P is infinite it must contain a diverging computation ❞



Dijkstra's objections to the operational argument

Dijkstra exploited the operational argument against unbounded nondeterminism in a preliminary version of Chapter 9 of "A Discipline of Programming".

He discarded it for two reasons:

- ❝ I want to ignore that my program texts also admit the interpretation of executable code ❞
- In the absence of an adequacy proof (every final state prescribed by the model can be reached by a computation), the finiteness of the set of final states

  ❝ could be a property of the implementation — namely, that it can only realize a finite number of the infinitely many permissible states ❞

Actor computation

Actors and their semantics

Unbounded nondeterminism was much discussed in the circle of students of Carl Hewitt at CSAIL working on the actor model of computation in the mid 1970s.

- Actors are deterministic computational agents that communicate by sending messages.
- Each message sent is guaranteed to arrive at its target actor, waiting in the actor's local queue to be processed.
- The arrival order is nondeterministic.
- The processing of a message may involve:
    - sending new messages;
    - changing local state.

Actors are a pioneering model of computation based on a notion of causal ordering on distributed events.
There are laws on event ordering excluding non-computable behaviors (e.g.: Zeno machines, Huffman's lamps, accelerating Turing machines, etc.).

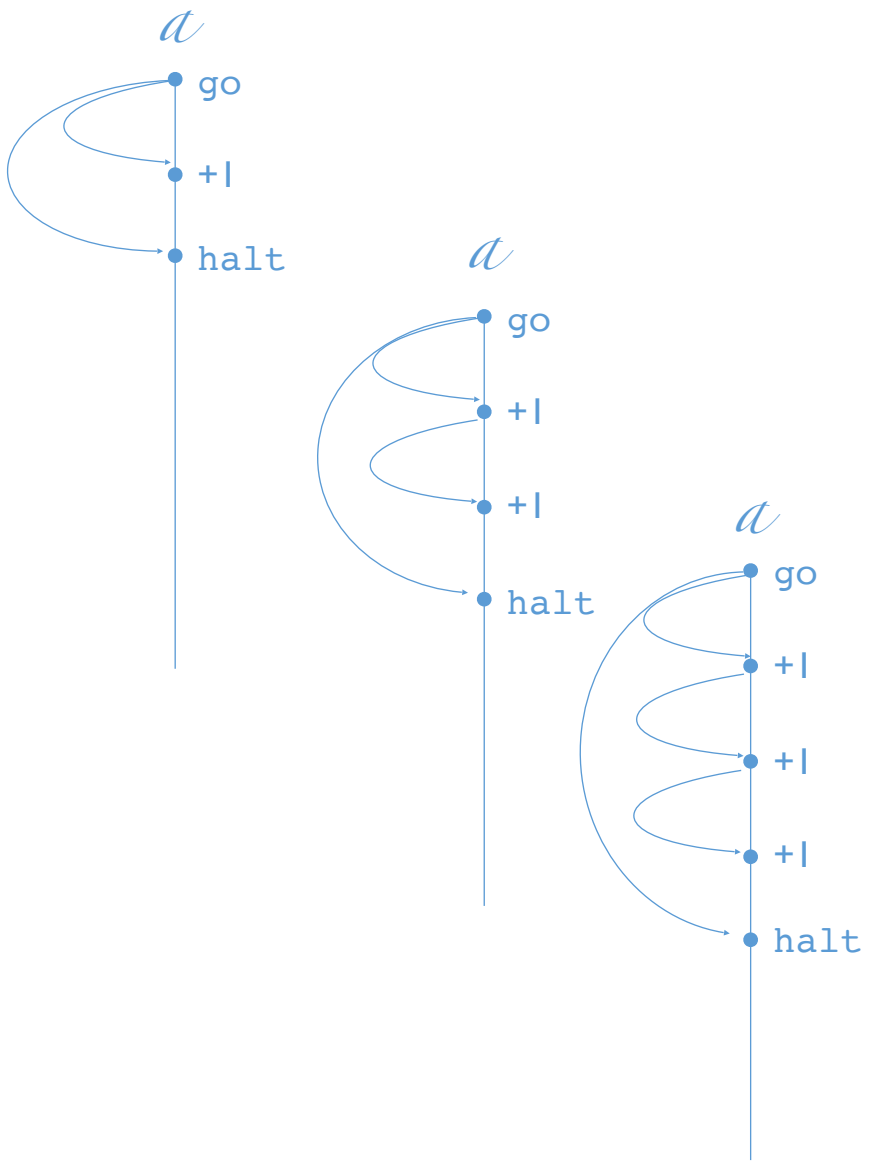
Hewitt's and Clinger's case for unbounded nondeterminism

Terminating unbounded choice

Consider an actor a which, upon initialization, sends itself

- a `halt` message, as well as
- `increment` messages.

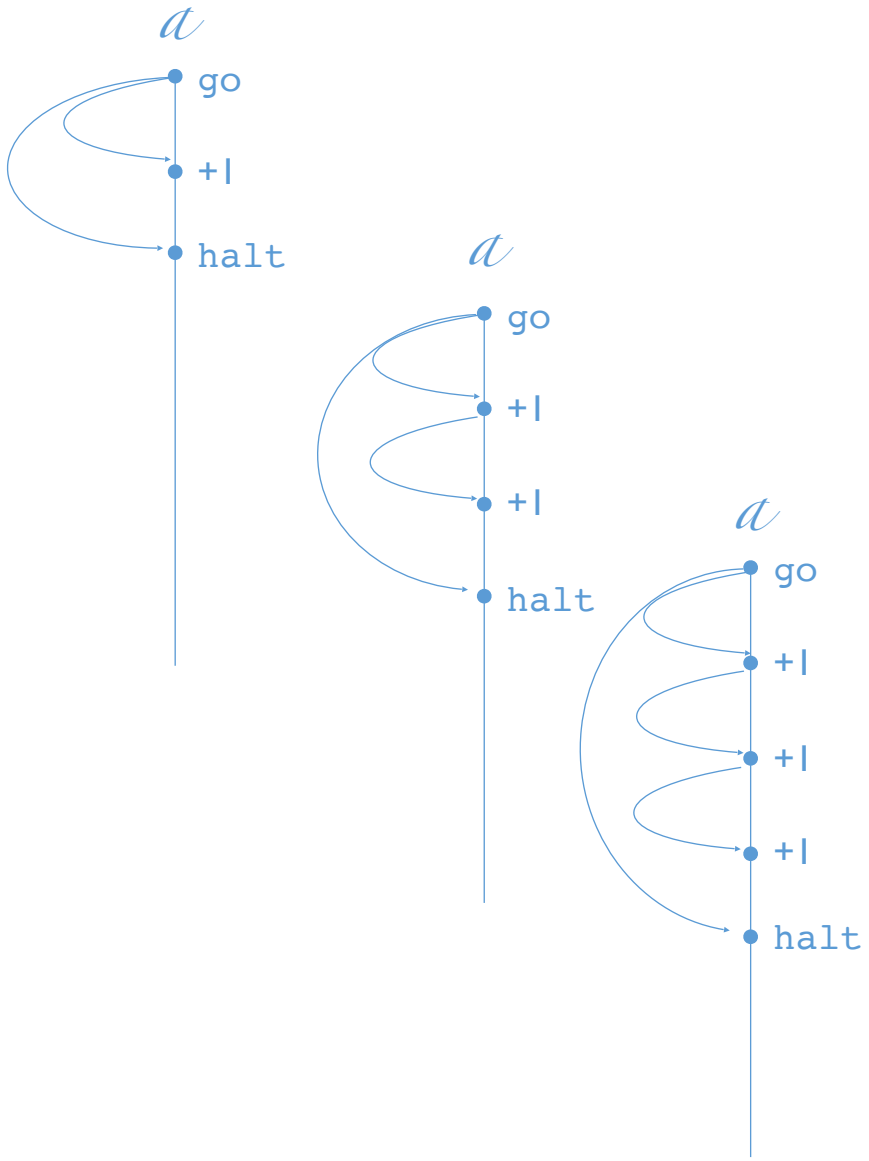
Since all messages sent eventually arrive at their targets, a will eventually receive this `halt` message and terminate, with an unbounded but finite value.



Hewitt's and Clinger's case for unbounded nondeterminism

Terminating unbounded choice

❝ This actor system has unbounded nondeterminism, yet it always halts.
It is clear that the longer computations are not very likely to happen in any reasonable implementation.
It is difficult to see how an implementation could guarantee termination without putting a bound on the nondeterminism, but implementations are not required to preserve all the nondeterminacy present in the semantics ❞ (Clinger)



The objection to the operational argument

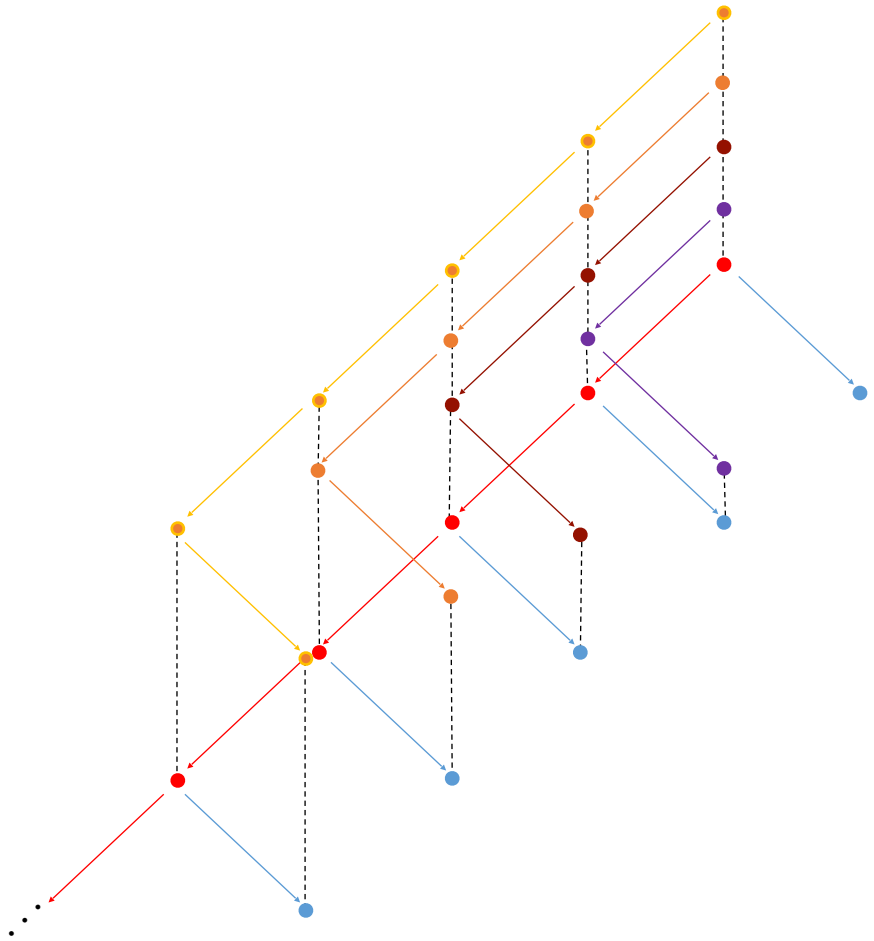
❝ The proof depends upon the premise that

if every node x of a certain infinite branch can be reached by some computation c,
then there exists a computation c that goes through every node x on the branch

This premise follows not from logic but from the interpretation given to choice points ❞ (Clinger 1981)

Finite delay allows the infinitely many terminating colored transition sequences, but **not** the infinite red one.

Nondeterminism is taken in the loose sense.



The status of the finite delay property

Microscopic causes of unbounded finite delay

Arbiter: a device that makes a discrete decision based on a continuous range of values.

Fundamental result on arbiters: It is impossible to build an arbiter that always decides within a bounded length of time.

The problem is due to meta-stable states ("the glitch"). Most proofs (e.g., Lamport's) use continuous models. Anderson & Gouda prove this for a discrete model, but in the proof unbounded = infinite

- ☛ König's Lemma strikes back

Macroscopic causes of finite unbounded delay

Routing over a network. Server crash. Failures.

Here finite unbounded delay is the assumption that messages are not lost. But how to guarantee such property?

The fairness issue

Dijkstra

❝ Call an obligation void if it is impossible to detect if it has not been fulfilled ❞

```
    b := true
; do b → print(0) ▯ b → print(1); b := false   od
```

it can be implemented by :

```
b := true
; do b → print(1); b := false  od
```

or even by:

```
b := true
; do b → print(0) od
```

Moral:

❝ void obligations should not occur in contracts, fairness […] can be ignored with impunity ❞

The fairness issue

Lamport & Schneider

❝ anyone who accepts the argument of [Dijkstra], that fairness can be ignored, must also be prepared to ignore termination and all other liveness properties ❞

Chandy & Misra

❝ Finite experiments cannot distinguish between fair and unfair implementations. So fairness has no place (Prof. Dijkstra argues) in program design.

Mathematicians and computer scientists often introduce concepts that cannot be verified by experiments, have no analog in the real world, and are patently unimplementable. These assumptions are made not because they are "correct" but because they allow programmers to separate concerns […]

We view fairness as a simplifying assumption. The important question to ask is not if fairness is real, but does it help? ❞

Properties of transitions systems

- Finitely observable properties of an LTS are those P such that,

  if α in S^ω has P, there is a finite initial segment w of α such that
  wβ has P for every β in S^ω

  Such properties are open subsets of the suitable Baire space.

- Property P is irrefutable for α in S^ω if it cannot be refuted along α:

  for every finite prefix w of α there is β in S^ω such that wβ has P

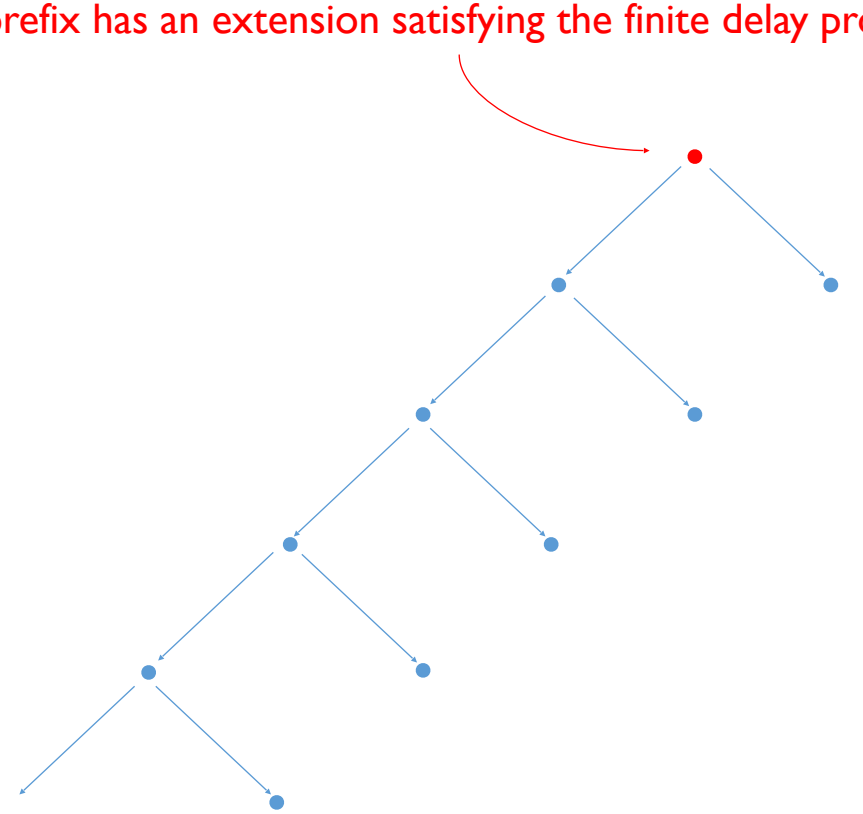
  In this case α is a limit point of P. Liveness properties (Alpern & Schneider) are irrefutable for any α in S^ω. (Examples: termination; finite delay)
  They are topologically dense.

(cfr. Kevin Kelly, *The Logic of Reliable Inquiry*, 1996)

Irrefutability

A property, like e.g. finite delay, may be irrefutable for α but false of α.

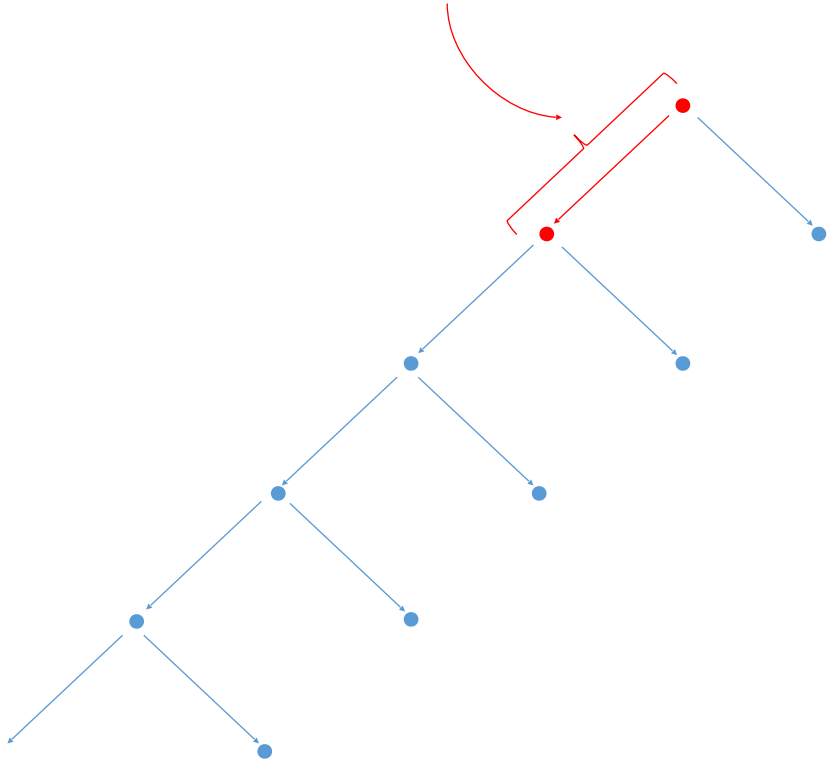
Example



Irrefutability

A property, like e.g. finite delay, may be irrefutable for α but false of α.

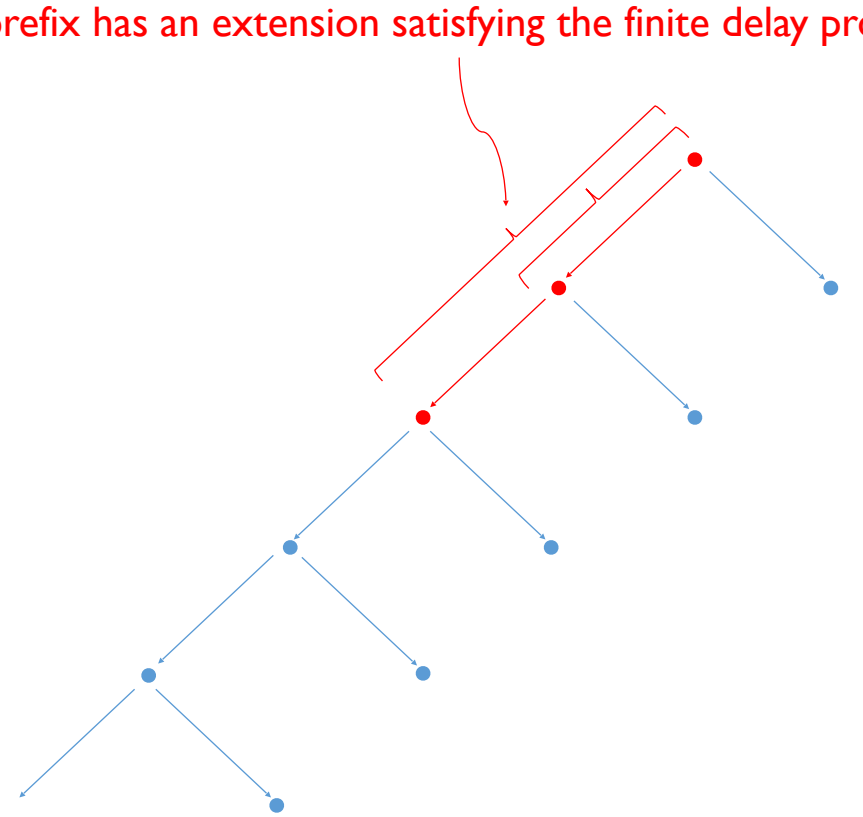
Example



Irrefutability

A property, like e.g. finite delay, may be irrefutable for α but false of α.

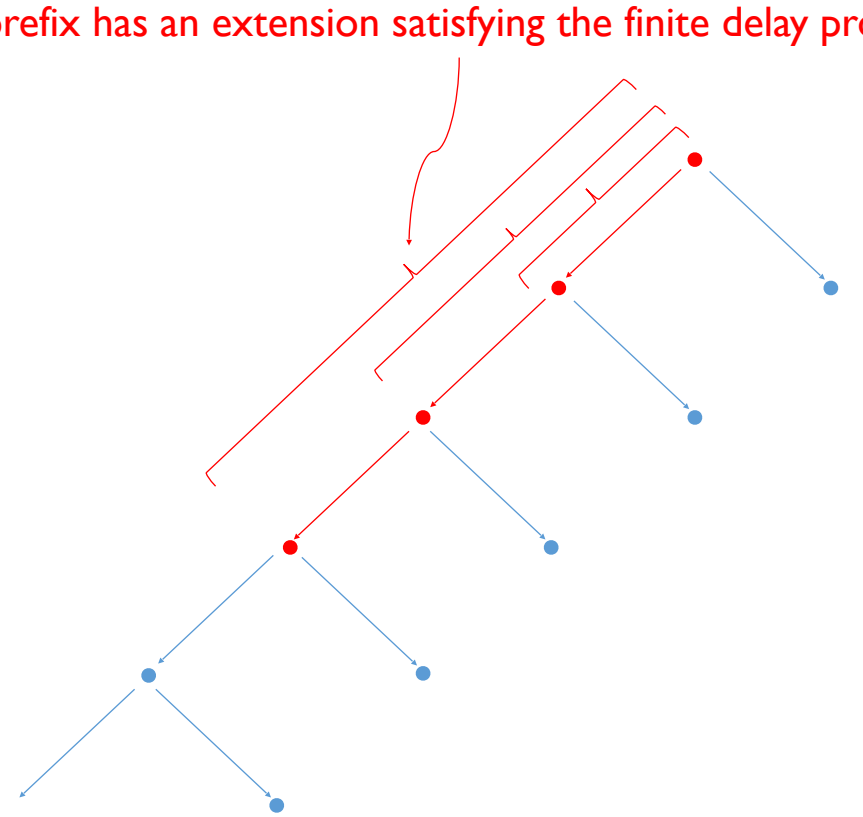
Example



Irrefutability

A property, like e.g. finite delay, may be irrefutable for α but false of α.

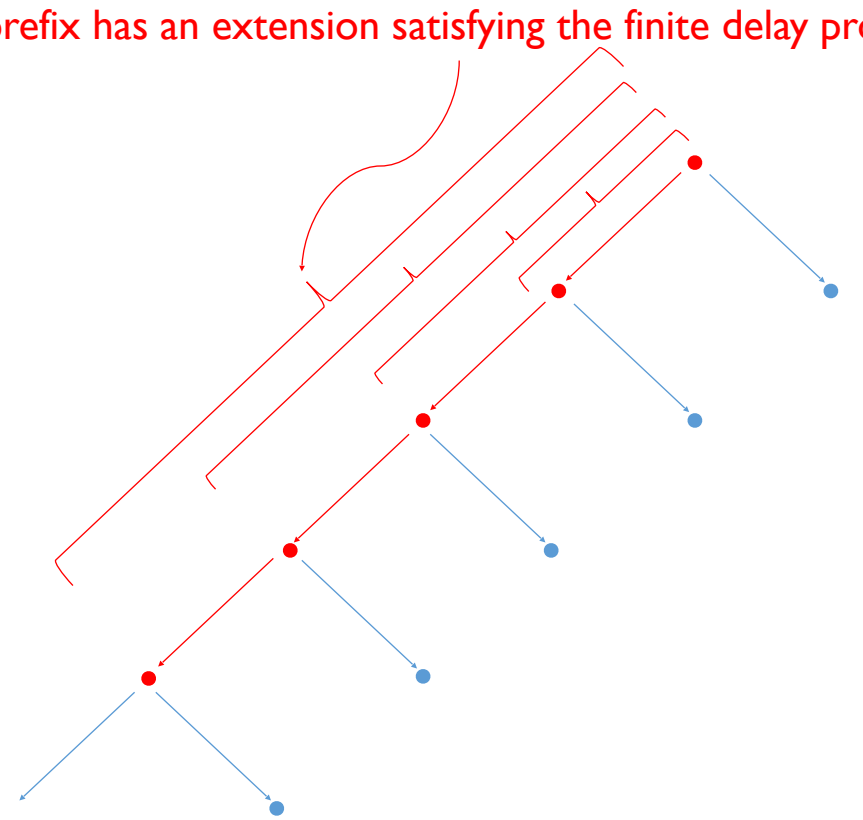
Example



Irrefutability

A property, like e.g. finite delay, may be irrefutable for α but false of α.

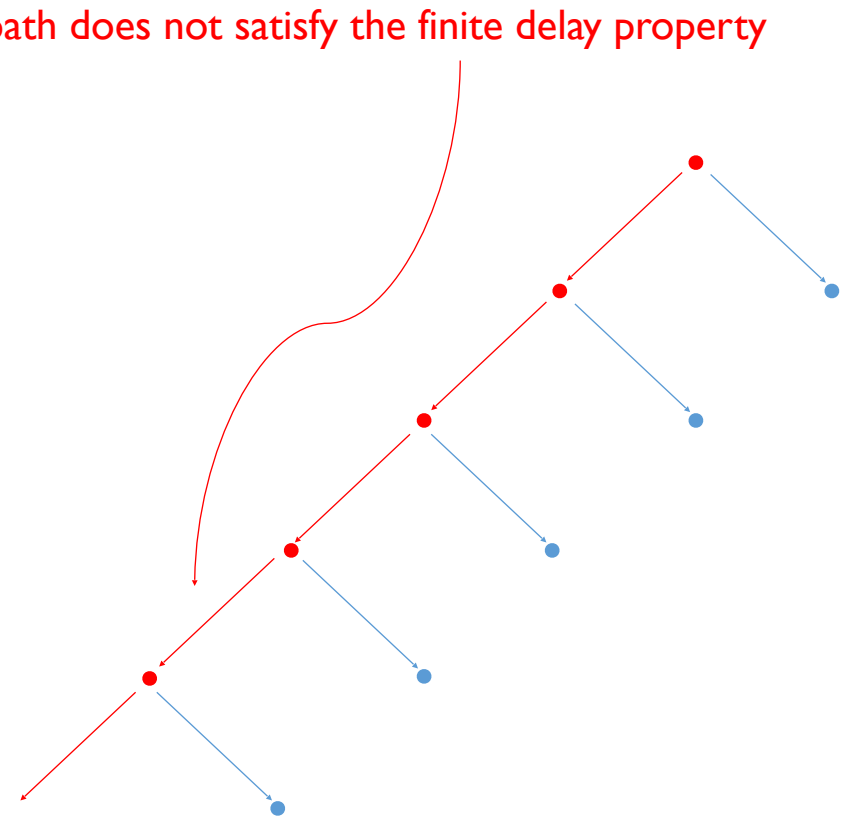
Example



Irrefutability

A property, like e.g. finite delay, may be irrefutable for α but false of α.

Example



Truth and irrefutable properties

Irrefutable properties cannot be ignored in the design of systems.
Their truth can give a system a different structure.

Examples

"I promise to give you back your money" is irrefutable. What could be the structure of a system based on such promises?

We normally make promises refutable:

"I promise to give you back your money in a week".

Truth and irrefutable properties

Irrefutable properties cannot be ignored in the design of systems.
Their truth can give a system a different structure.

Examples

In systems where messages reach their destination eventually, synchronization can be achieved via request/acknowledge cycles.

Failure of this irrefutable property jeopardizes this signaling scheme.

Truth and irrefutable properties

Irrefutable properties cannot be ignored in the design of systems.
Their truth can give a system a different structure.

Examples

When termination is a necessary consequence of rich type disciplines where programs meet their specifications (= inhabit types) by construction.

Failure of termination makes correctness-by-construction inapplicable.

1. introduction

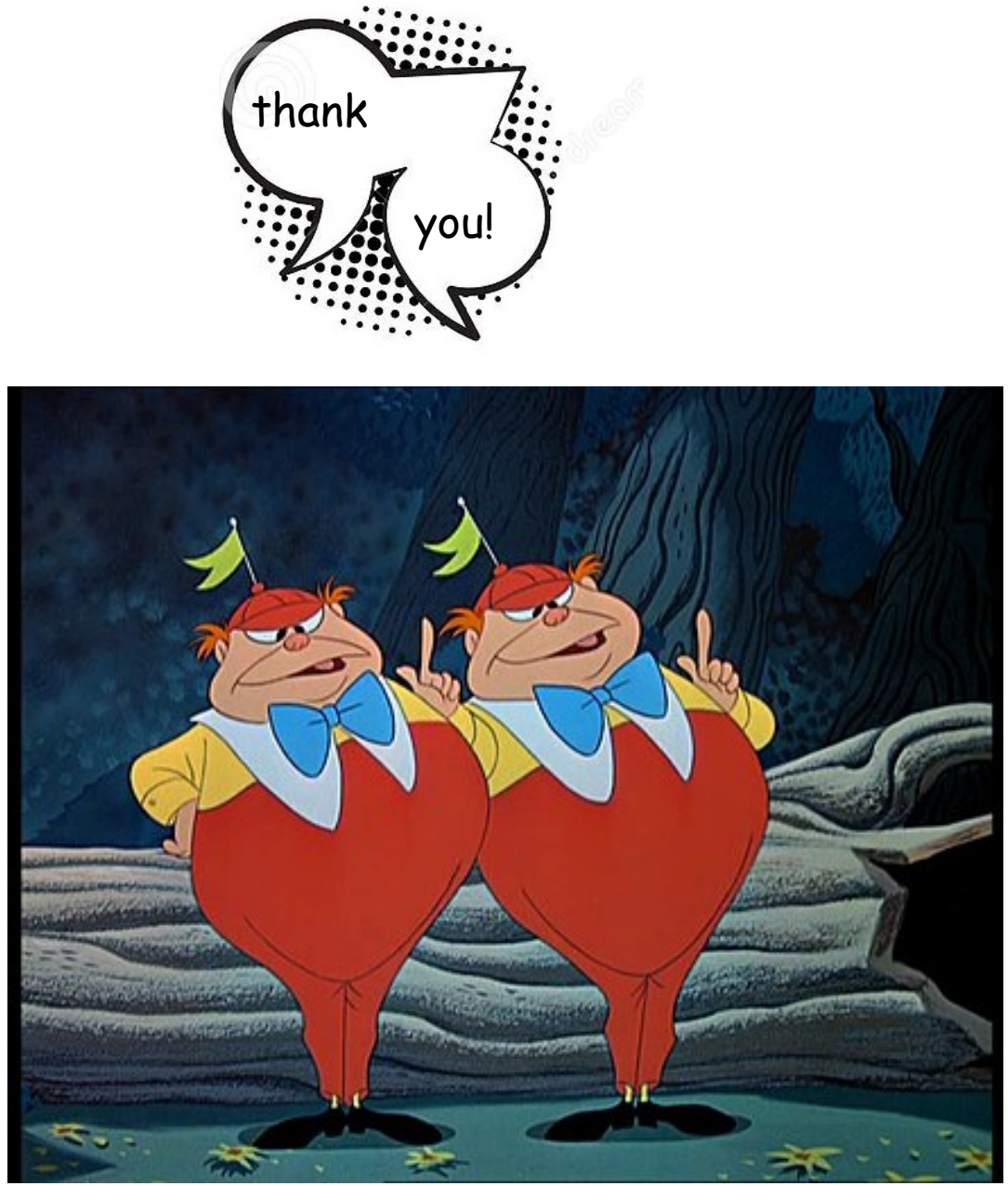
2. technical survey

3. retrospective

Philosophical relevance of all this

Why unbounded nondeterminism should be interesting to the philosopher of computing?

- A possible source of hypercomputation;
- A laboratory for dissecting models of computational phenomena;
- A chance to lay bare the relations between semantics and implementation of programs.


thank
you!

Further topics

Interpreting McCarthy's amb

For an expression a and a value v, let a $\rightarrow_I$ v mean that v is a value of a.
Dovetailing amb(a,b) of the computations of a and b has the interpretation

amb(a,b) $\rightarrow_I$ v if and only if a $\rightarrow_I$ v or b $\rightarrow_I$ v

If computation can be broken into atomic steps, so that a $\rightarrow_{In}$ v means that a computes to v in at most n steps, and a $\rightarrow_I$ v if and only if a $\rightarrow_{In}$ v for some n, and a $\rightarrow_{In}$ means that a converges in at most n steps, we can define dovetailing:

amb(a,b) = d(0) **where** d(n) = **if not** a $\rightarrow_{In}$ **then** **if not** b $\rightarrow_{In}$ **then** d(n+1) **else** b **else** a

Observe that this implementation of amb(a,b) converges iff at least one of a,b converges.

Nondeterminism and parallelism

The parallel-or issue

p-or	T	F	⊥
T	**T**	**T**	T
F	**T**	**F**	⊥
⊥	T	⊥	⊥

Remark: p-or(x,y) = amb(**if** x **then** true **else** y,
if y **then** true **else** x)

There is no argument place that needs to be visited in order to get the result of p-or(x,y):
p-or is not sequential (Vuillemin 1973), nor stable (Berry 1976)