Unique solution techniques for processes and functions

Adrien Durier

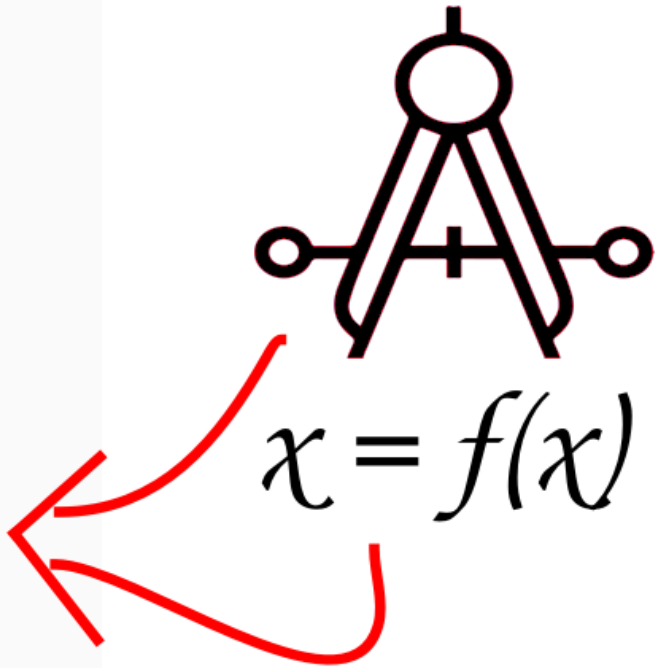
Cotutelle ENS de Lyon & Università di Bologna

on the web
11th June, 2020

Formal verification of software

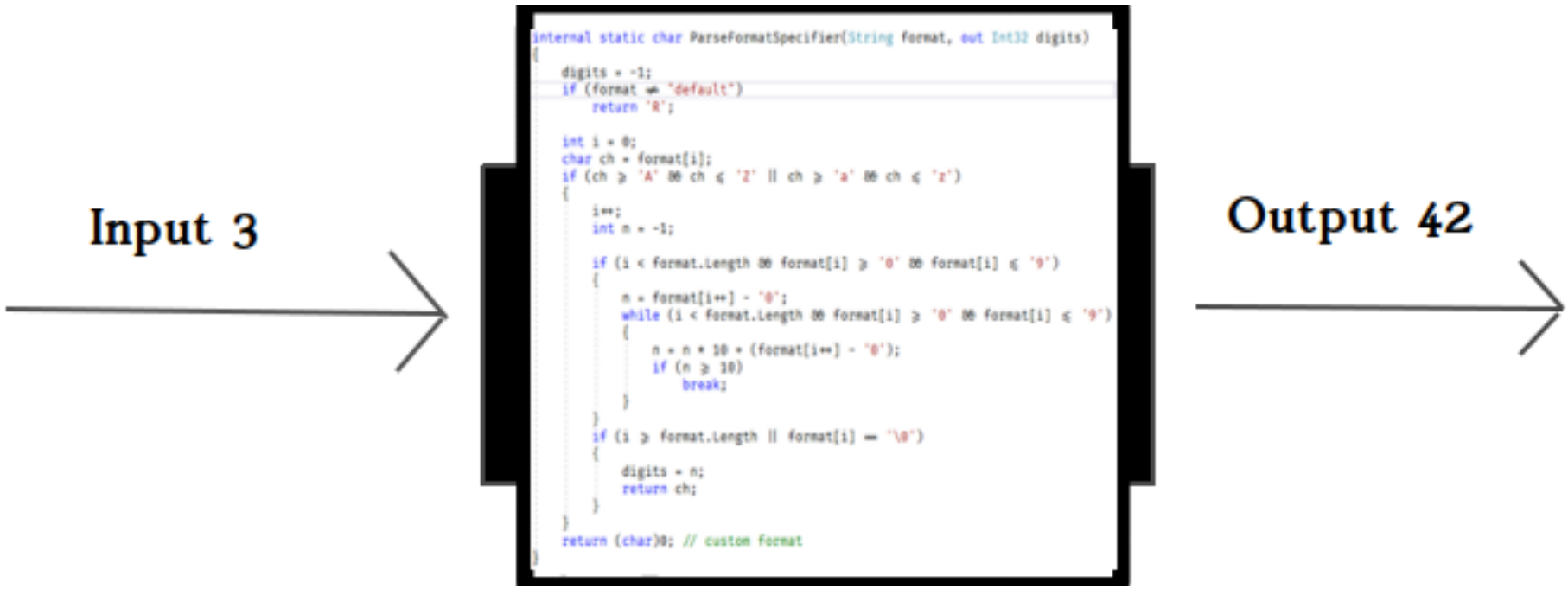
Formal verification of software

```
type expr =
    | TInt of int                    (* i *)
    | TAdd of expr * expr            (* e1+e2 *)
    | TMul of expr * expr            (* e1*e2 *)
    | TDiv of expr * expr            (* e1/e2 *)


let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = TDiv(t1,TDiv(t1, t1))
let t3 = TDiv(t1, TInt 0)

exception Div_by_Zero

let eval : expr -> int = fun e -> 0

(* tests *)
let _ = eval t1
let _ = eval t2
let _ = try eval t3 with Div_by_Zero -> 55

let eval_count : expr -> int = fun e -> 0

(* tests *)
let _ = eval_count t1
let _ = eval_count t2
let _ = try eval_count t3 with Div_by_Zero -> 55

type expr =
    | TInt of int                    (* i *)
    | TAdd of expr * expr            (* e1+e2 *)
    | TMul of expr * expr            (* e1*e2 *)
    | TDiv of expr * expr            (* e1/e2 *)
    | TLet of string * expr * expr   (* let x = e1 in e2 *)
    | TVar of string                 (* x *)

(* Exemples d'expressions : *)
let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
```

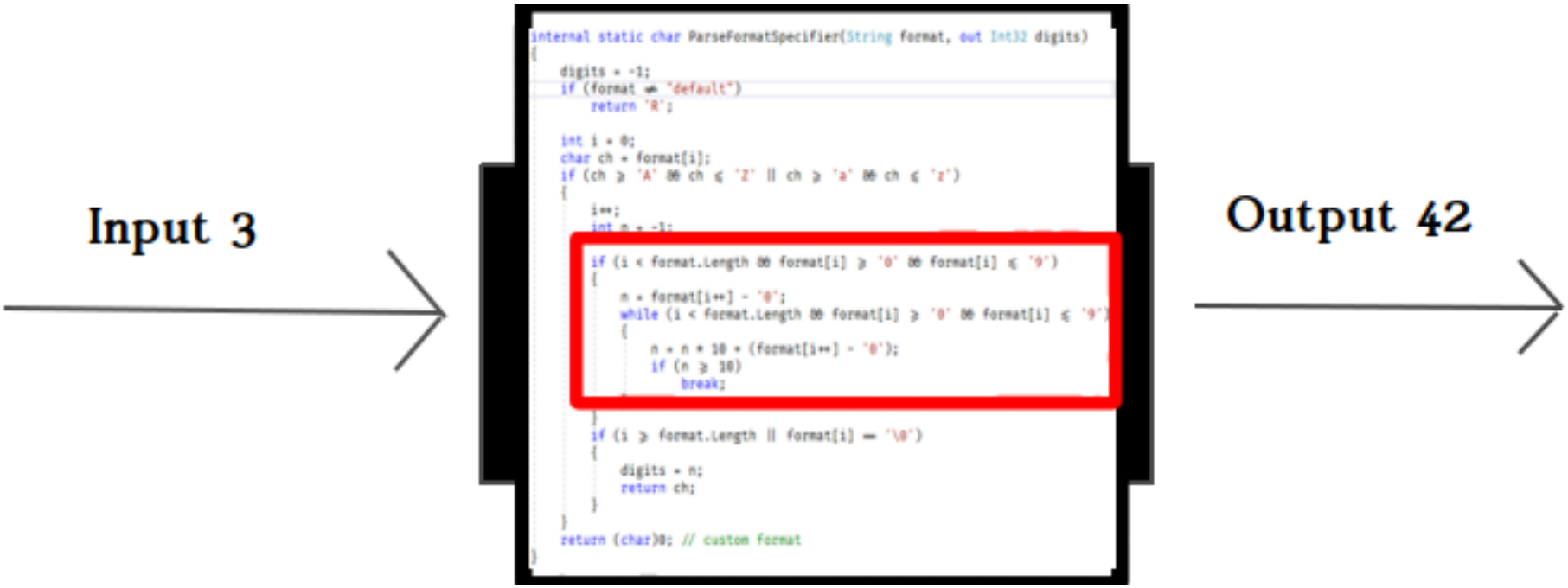


Programs as Black Boxes

Programs as Black Boxes



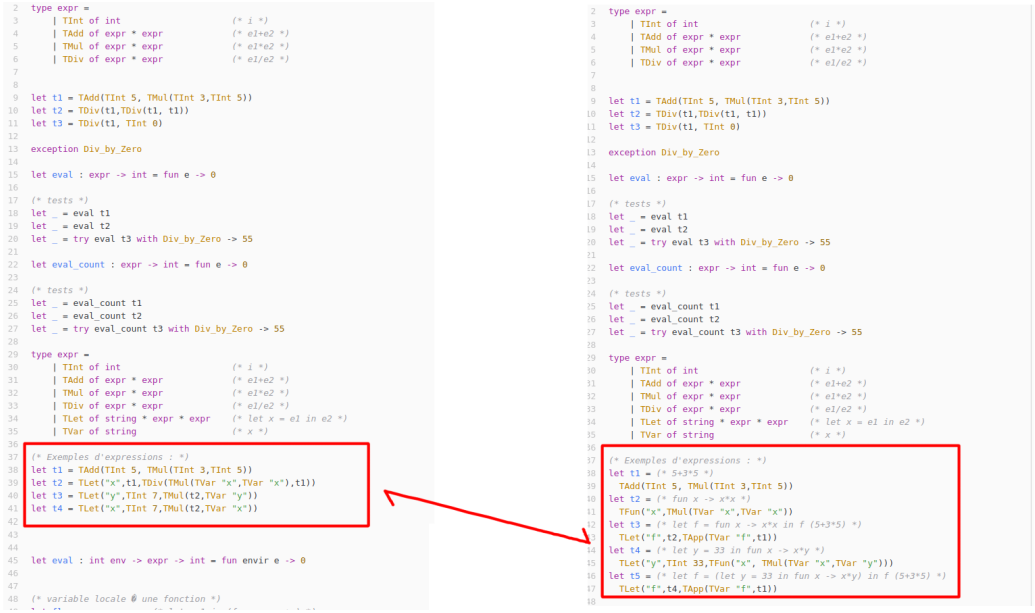
Programs as Black Boxes



Program equivalence

Context: Formal study of programming

Prove **program equivalences**

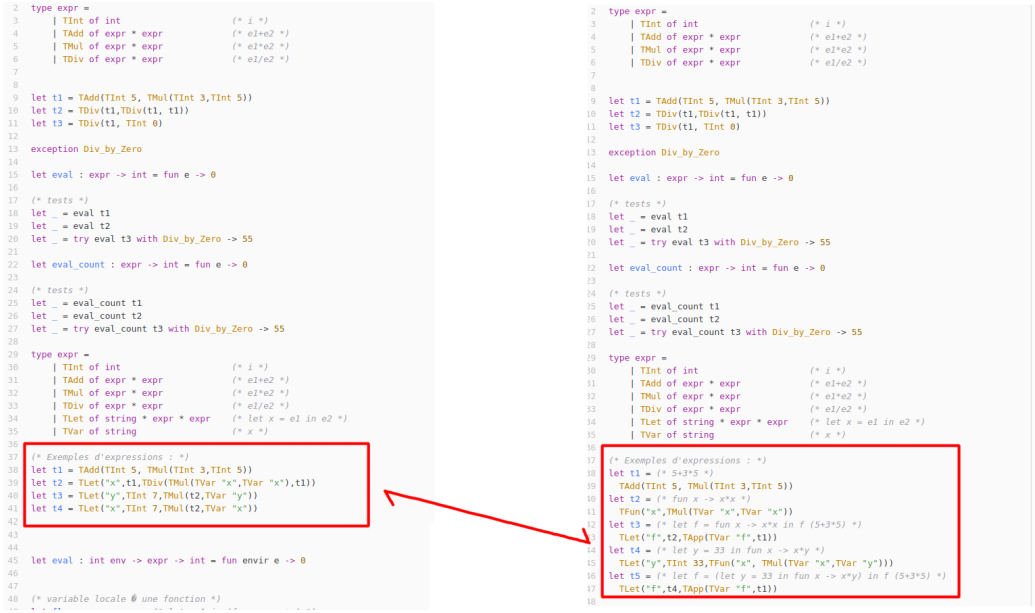


```
type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)


let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = TDiv(t1,TDiv(t1, t1))
let t3 = TDiv(t1, TInt 0)

exception Div_by_Zero

let eval : expr -> int = fun e -> 0

(* tests *)
let _ = eval t1
let _ = eval t2
let _ = try eval t3 with Div_by_Zero -> 55

let eval_count : expr -> int = fun e -> 0

(* tests *)
let _ = eval_count t1
let _ = eval_count t2
let _ = try eval_count t3 with Div_by_Zero -> 55

type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)
    | TLet of string * expr * expr  (* let x = e1 in e2 *)
    | TVar of string                (* x *)

(* Exemples d'expressions : *)
let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = TLet("x",t1,TDiv(TMul(TVar "x",TVar "x"),t1))
let t3 = TLet("y",TInt 7,TMul(t2,TVar "y"))
let t4 = TLet("x",TInt 7,TMul(t2,TVar "x"))


let eval : int env -> expr -> int = fun envir e -> 0


(* variable locale à une fonction *)
```

```
type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)


let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = TDiv(t1,TDiv(t1, t1))
let t3 = TDiv(t1, TInt 0)

exception Div_by_Zero

let eval : expr -> int = fun e -> 0

(* tests *)
let _ = eval t1
let _ = eval t2
let _ = try eval t3 with Div_by_Zero -> 55

let eval_count : expr -> int = fun e -> 0

(* tests *)
let _ = eval_count t1
let _ = eval_count t2
let _ = try eval_count t3 with Div_by_Zero -> 55

type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)
    | TLet of string * expr * expr  (* let x = e1 in e2 *)
    | TVar of string                (* x *)

(* Exemples d'expressions : *)
let t1 = (* 5+3*5 *)
  TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = (* fun x -> x*x *)
  TFun("x",TMul(TVar "x",TVar "x"))
let t3 = (* let f = fun x -> x*x in f (5+3*5) *)
  TLet("f",t2,TApp(TVar "f",t1))
let t4 = (* let y = 33 in fun x -> x*y *)
  TLet("y",TInt 33,TFun("x", TMul(TVar "x",TVar "y")))
let t5 = (* let f = (let y = 33 in fun x -> x*y) in f (5+3*5) *)
  TLet("f",t4,TApp(TVar "f",t1))
```

- **Goal:** verify an optimization (for example)
- A mathematical proof that two programs can be substituted for each other

Program equivalence



```
type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)


let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = TDiv(t1,TDiv(t1, t1))
let t3 = TDiv(t1, TInt 0)

exception Div_by_Zero

let eval : expr -> int = fun e -> 0

(* tests *)
let _ = eval t1
let _ = eval t2
let _ = try eval t3 with Div_by_Zero -> 55

let eval_count : expr -> int = fun e -> 0

(* tests *)
let _ = eval_count t1
let _ = eval_count t2
let _ = try eval_count t3 with Div_by_Zero -> 55

type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)
    | TLet of string * expr * expr  (* let x = e1 in e2 *)
    | TVar of string                (* x *)

(* Exemples d'expressions : *)
let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = TLet("x",t1,TDiv(TMul(TVar "x",TVar "x"),t1))
let t3 = TLet("y",TInt 7,TMul(t2,TVar "y"))
let t4 = TLet("x",TInt 7,TMul(t2,TVar "x"))


let eval : int env -> expr -> int = fun envir e -> 0


(* variable locale à une fonction *)
```

```
type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)


let t1 = TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = TDiv(t1,TDiv(t1, t1))
let t3 = TDiv(t1, TInt 0)

exception Div_by_Zero

let eval : expr -> int = fun e -> 0

(* tests *)
let _ = eval t1
let _ = eval t2
let _ = try eval t3 with Div_by_Zero -> 55

let eval_count : expr -> int = fun e -> 0

(* tests *)
let _ = eval_count t1
let _ = eval_count t2
let _ = try eval_count t3 with Div_by_Zero -> 55

type expr =
    | TInt of int                   (* i *)
    | TAdd of expr * expr           (* e1+e2 *)
    | TMul of expr * expr           (* e1*e2 *)
    | TDiv of expr * expr           (* e1/e2 *)
    | TLet of string * expr * expr  (* let x = e1 in e2 *)
    | TVar of string                (* x *)

(* Exemples d'expressions : *)
let t1 = (* 5+3*5 *)
  TAdd(TInt 5, TMul(TInt 3,TInt 5))
let t2 = (* fun x -> x*x *)
  TFun("x",TMul(TVar "x",TVar "x"))
let t3 = (* let f = fun x -> x*x in f (5+3*5) *)
  TLet("f",t2,TApp(TVar "f",t1))
let t4 = (* let y = 33 in fun x -> x*y *)
  TLet("y",TInt 33,TFun("x", TMul(TVar "x",TVar "y")))
let t5 = (* let f = (let y = 33 in fun x -> x*y) in f (5+3*5) *)
  TLet("f",t4,TApp(TVar "f",t1))
```

- Equivalent programs: produce the same result, etc. . .
- Partial programs : x might be used while undefined (defined in another part of the program)
- Indistinguishable in any context
  - → can modify the rest of the code in the same way
- **Context:** Program with a hole $[\cdot]$. The hole can be replace by a partial program.

Program equivalence

Contexts

If context C is:

```
int x;
[·]
x = x++;
```

If the partial program **P** is:

```
x = 3;
```

Then $C[\mathbf{P}]$ is the program:

```
int x;
x = 3;
x = x++;
```

by a partial program.

Contextual equivalence

Contextual equivalence defines which pairs of programs can be freely substituted.
If **P** and **Q** are **contextually equivalent** (**P** $\simeq$ **Q**), then they are interchangeable with each other.

$$\mathbf{P} \simeq \mathbf{Q} \stackrel{\text{def}}{=} \text{for all contexts } C,$$
$$C[\mathbf{P}] \text{ et } C[\mathbf{Q}] \text{ produce the same } \textbf{values}.$$
$$(\text{have the same } \textbf{observables})$$

- Universal quantifier **"for all"**
- **Proof techniques** are essential
  - Set of **sufficient** conditions on **P** and **Q** to ensure **P** $\simeq$ **Q**

Contributions

I **Full Abstraction** for
Milner's encoding of the call-by-value λ-calculus in the π-calculus

II A **fixpoint theorem** for weak behavioural equivalences

The call-by-value λ-calculus

Open call-by-value

Open *call-by-value*

Call-by-value β-reduction $(\lambda x.M)V \rightarrow_{\beta_v} M\{V/x\}$

Open *call-by-value*

Call-by-value β-reduction $\quad (\lambda x.M)V \to_{\beta_v} M\{V/x\}$

Open terms

Values	$V \;:=\; x \;\mid\; \lambda x.M$
Stuck redexes	$(\lambda z.z)(xV) \not\to_{\beta_v} xV$
Normal forms	$V \;\mid\; C_{\mathrm{e}}[xV]$

Open *call-by-value*

Call-by-value β-reduction $\quad (\lambda x.M)V \to_{\beta_v} M\{V/x\}$

Open terms

Values	$V \;:=\; x \;\mid\; \lambda x.M$
Stuck redexes	$(\lambda z.z)(xV) \not\to_{\beta_v} xV$
Normal forms	$V \;\mid\; C_{\mathrm{e}}[xV]$

Evaluation contexts	$C_{\mathrm{e}} \;:=\; [\cdot] \;\mid\; C_{\mathrm{e}}M \;\mid\; VC_{\mathrm{e}}$
β_v reduction	$C_{\mathrm{e}}[(\lambda x.M)V] \to_{\beta_v} C_{\mathrm{e}}[M\{V/x\}]$

Eager Normal Form Bisimulation [Lassen05]

$\simeq_e$ stands for eager normal form bisimilarity

$M \simeq_e N$ whenever:

1. M and N diverge
2. $M \Rightarrow x$, $N \Rightarrow x$
3. $M \Rightarrow \lambda x.M'$, $N \Rightarrow \lambda x.N'$ and
$$M' \simeq_e N'$$
4. $M \Rightarrow C_e[xV]$, $N \Rightarrow C'_e[xV']$ and
$$\begin{cases} C_e[z] \simeq_e C'_e[z] \text{ for a fresh } z \\ V \simeq_e V' \end{cases}$$

Tree structures for *call-by-value*

$\simeq_e$ stands for eager normal form bisimilarity

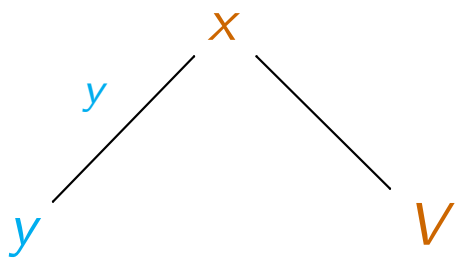
- $xV \simeq_e \mathrm{I}(xV)$
- $(\lambda z.\Omega)(xV) \not\simeq_e \Omega$
- $(\lambda z.xV)(xV) \not\simeq_e xV$

Tree structures for *call-by-value*

$\simeq_e$ stands for eager normal form bisimilarity

- $xV \simeq_e \mathrm{I}(xV)$

xV
(evaluation context: $[\cdot]$)



$\mathrm{I}(xV)$
$\mathrm{I}y \to y$

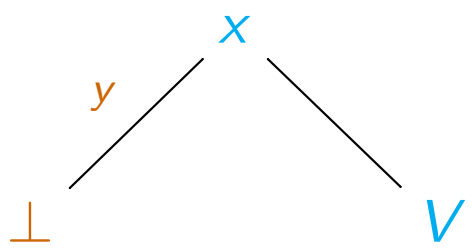
- $(\lambda z.\Omega)(xV) \not\simeq_e \Omega$
- $(\lambda z.xV)(xV) \not\simeq_e xV$

Tree structures for *call-by-value*

$\simeq_e$ stands for eager normal form bisimilarity

- $xV \simeq_e I(xV)$
- $(\lambda z.\Omega)(xV) \not\simeq_e \Omega$ (but are contextually equivalent)

$(\lambda z.\Omega)(xV)$



$\not\simeq_e$

Ω

$\bot$

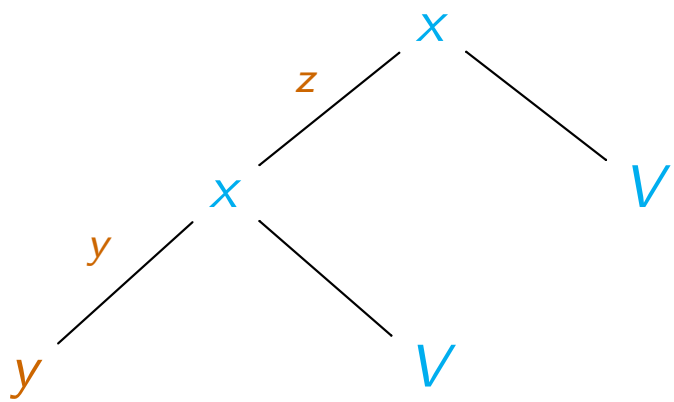
- $(\lambda z.xV)(xV) \not\simeq_e xV$

Tree structures for *call-by-value*

$\simeq_e$ stands for eager normal form bisimilarity

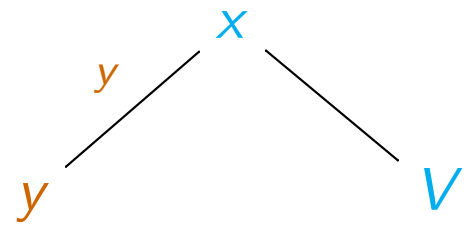
- $xV \simeq_e \mathrm{I}(xV)$
- $(\lambda z.\Omega)(xV) \not\simeq_e \Omega$
- $(\lambda z.xV)(xV) \not\simeq_e xV$ (but are contextually equivalent)

$(\lambda z.xV)(xV)$



$\not\simeq_e$

xV



Functions as concurrent processes

The untyped λ-calculus (1930s)

$\xrightarrow[\text{[Milner90]}]{[\![\cdot]\!]}$

The π-calculus (1989)

Functions as concurrent processes

The untyped λ-calculus (1930s)

$\xrightarrow[\text{[Milner90]}]{[\![\cdot]\!]}$

The π-calculus (1989)

Program execution: **interaction** between term and context

$\rightarrow$ we consider **open λ-terms**

Functions as concurrent processes

The untyped λ-calculus (1930s)

$\xrightarrow[\text{[Milner90]}]{[\![\cdot]\!]}$

The π-calculus (1989)

Program execution: **interaction** between term and context
$\rightarrow$ we consider **open λ-terms**

We study the problem of **Full Abstraction**:
Characterizing the **equivalence $\approx$ induced** by the encoding:

$$M \approx N \text{ iff } [\![M]\!] \simeq [\![N]\!]$$

Synchronization

The π-calculus: exchange of messages along **channels**

$(x, y, z \ldots \; p, q, r \ldots)$

Set of **channels**:

- Receive a channel name x on a channel $\mathtt{a}$: $\mathtt{a}(x)$
- Sending a channel name $\mathtt{b}$ on a channel $\mathtt{a}$: $\overline{\mathtt{a}}\langle \mathtt{b} \rangle$
- Special **internal action τ**

$$\mathtt{a}(x).P \mid \overline{\mathtt{a}}\langle b \rangle.Q \xrightarrow{\tau} P\{b/x\} \mid Q$$

- **Receiving** and **sending** on $\mathtt{a}$: possible **synchronization**
- A name is exchanged between the agents: an **internal action τ** happens
- τ is **invisible** to other agents

Interactive open programs

Programs as concurrent mobile interactive processes

I. **Full Abstraction** for
Milner's encoding of the call-by-value λ-calculus in the π-calculus

II. A new fixpoint theorem for weak behavioural equivalences

Milner's encoding

program execution is modelled as an **interaction** between

the program
(being evaluated)

the context
(in which it is evaluated)

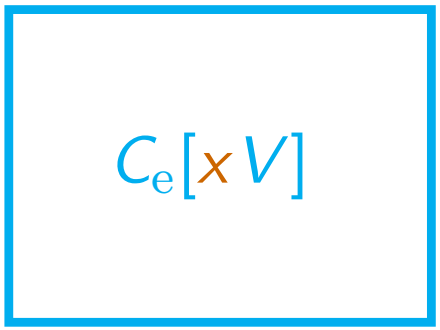
Evaluated arguments are stored by the context

The program requests them as needed

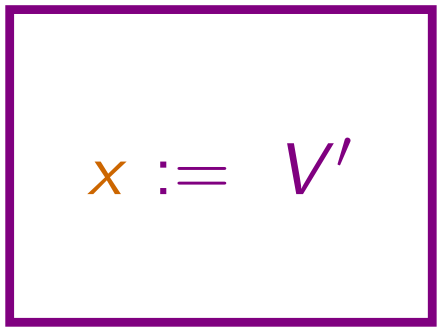
Milner's encoding

program execution is modelled as an **interaction** between

the program



the context



Milner's encoding

program execution is modelled as an **interaction** between

the program **the context**

Milner's encoding

program execution is modelled as an **interaction** between

the program $C_e[\cdot]$

the context $x := V'$

the evaluation of xV starts in a new location

$V'V$

Milner's encoding

program execution is modelled as an **interaction** between

the program $C_e[y]$

the context $x := V'$

once evaluated, it is stored in a new variable y

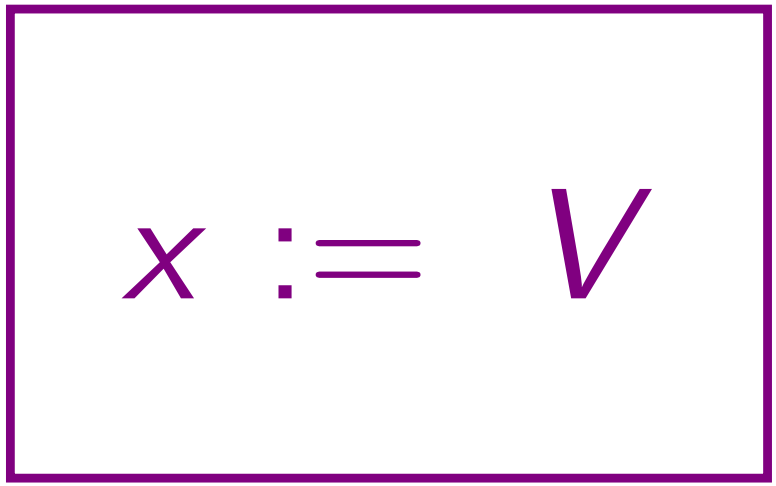
$y := V''$

Two types of processes and locations

program execution is modelled as an **interaction** between

the program
(being evaluated)

the context
(in which it is evaluated)



running term

evaluated term

Two types of processes and locations

program execution is modelled as an **interaction** between

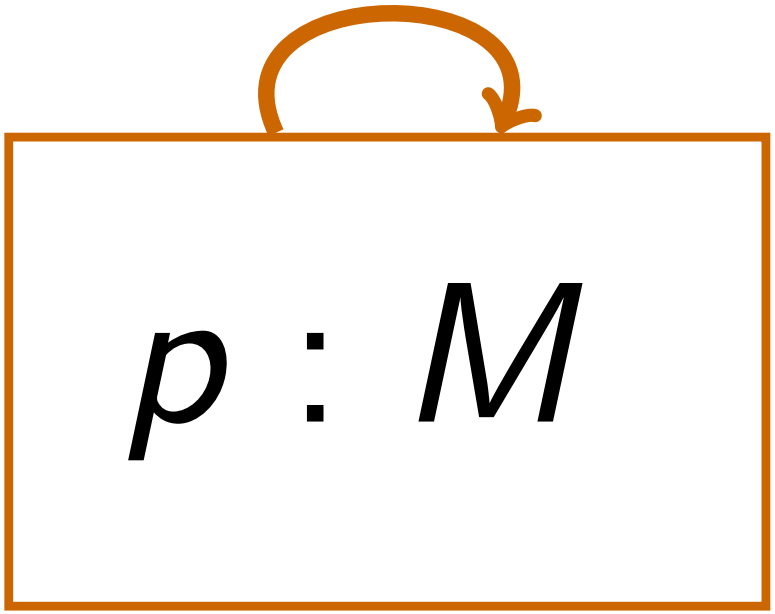
the program
(being evaluated)

the context
(in which it is evaluated)

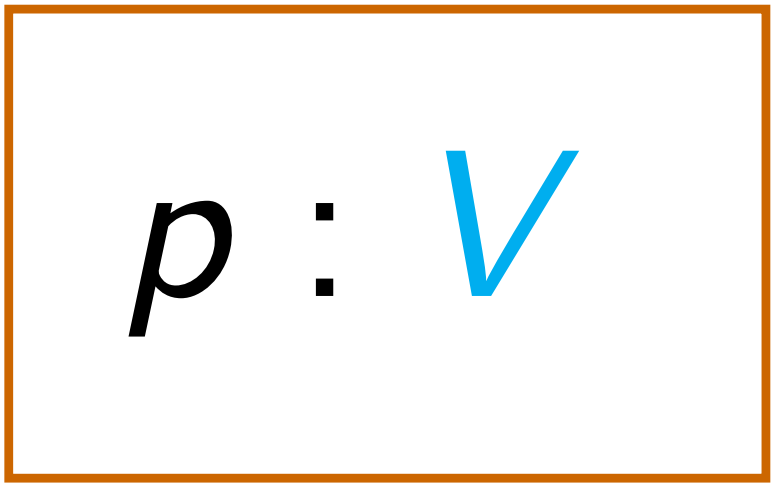
running term at location p
(or 'continuation' **p**)

evaluated term at variable x
(acts like an explicit substitution)

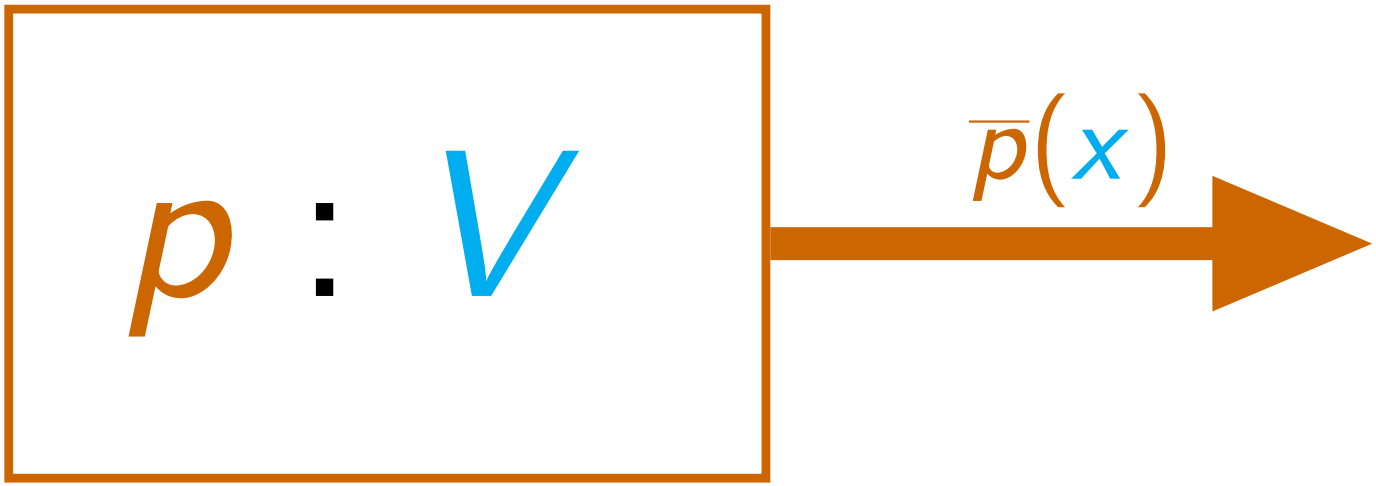
Evaluation of a term



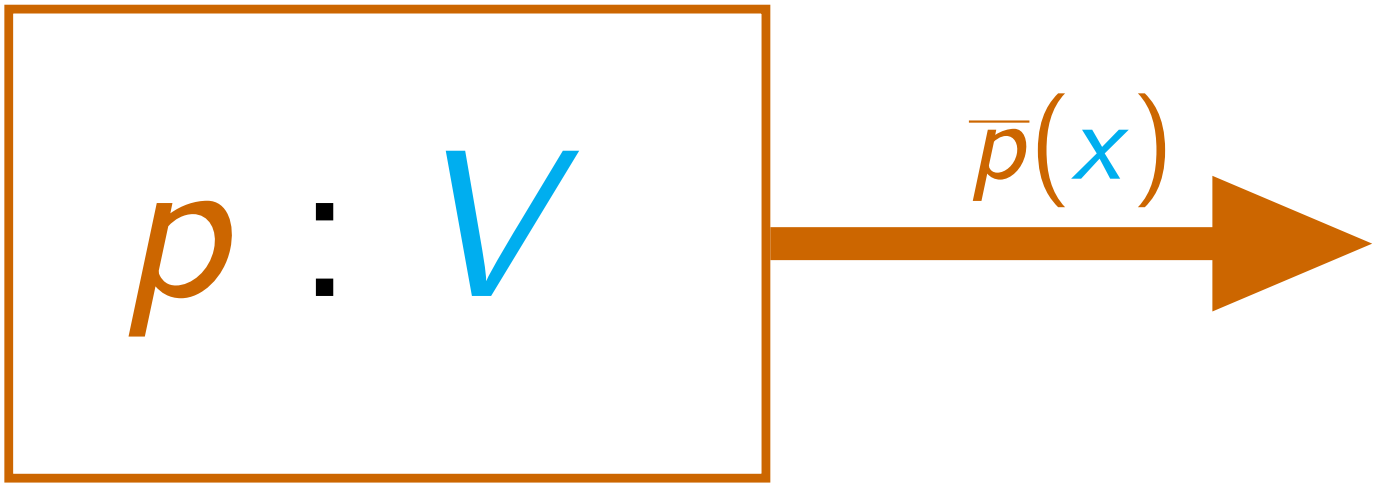
Evaluation of a term



Evaluation of a term



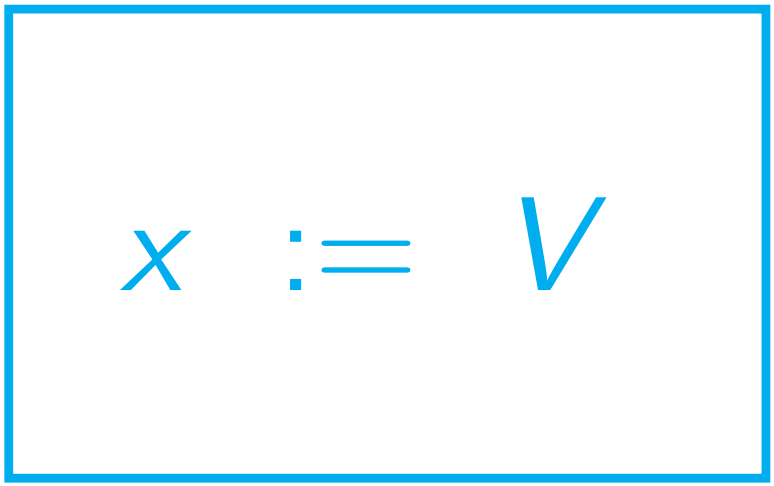
Evaluation of a term



$\overline{p}$: emits on p

x: new location (*fresh name*) for value V

Evaluation of a term

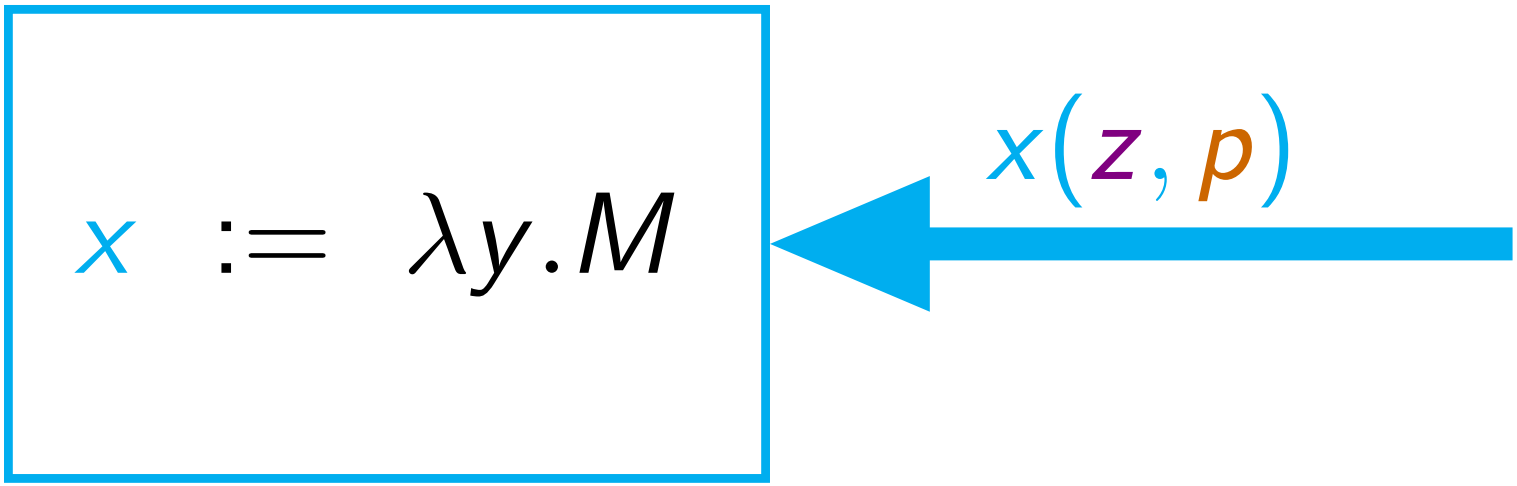


x: new location (*fresh name*) for value V

Function call

$$x := \lambda y.M$$

Function call

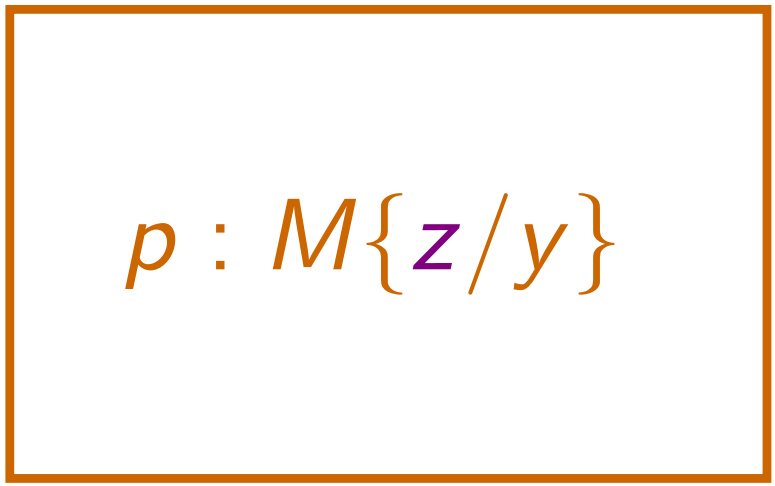


x: Address of the value being called

z: Address of argument

p: Evaluate at p

Function call



x: Address of the value being called

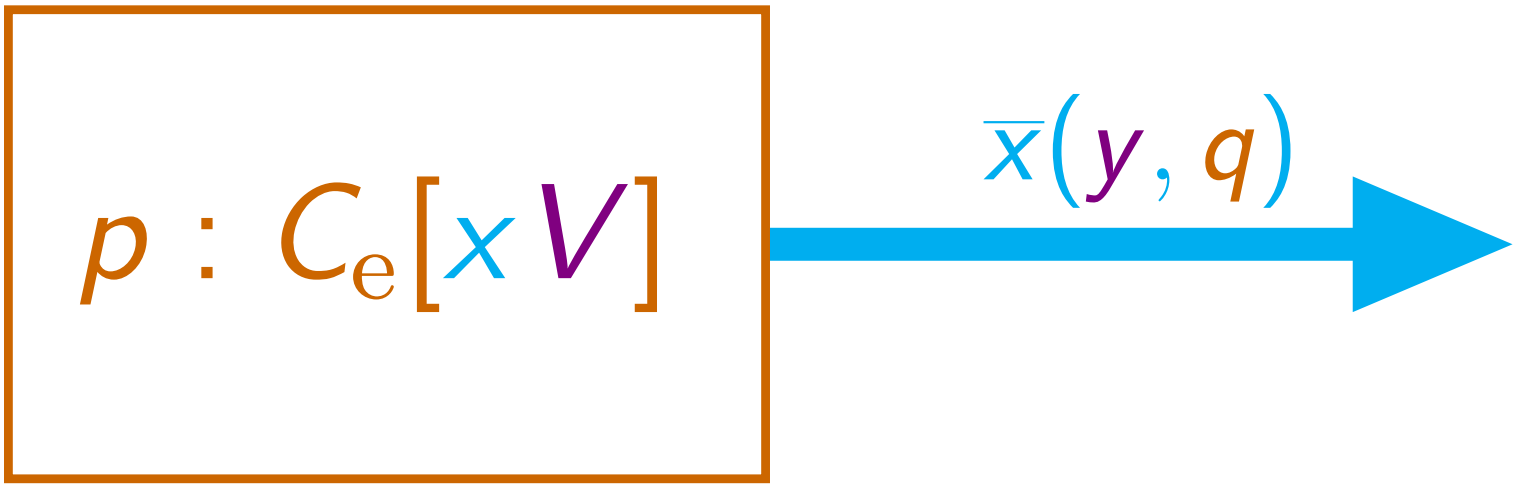
z: Address of argument

p: Evaluate at p

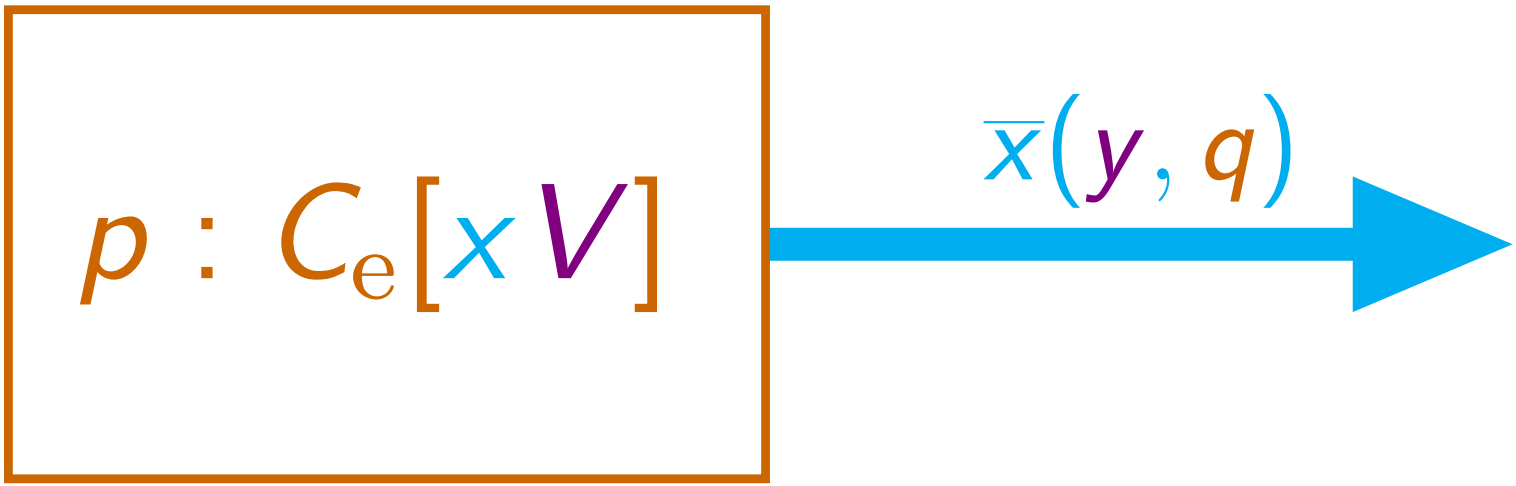
Calling a function

$$p : C_e[x\, V]$$

Calling a function



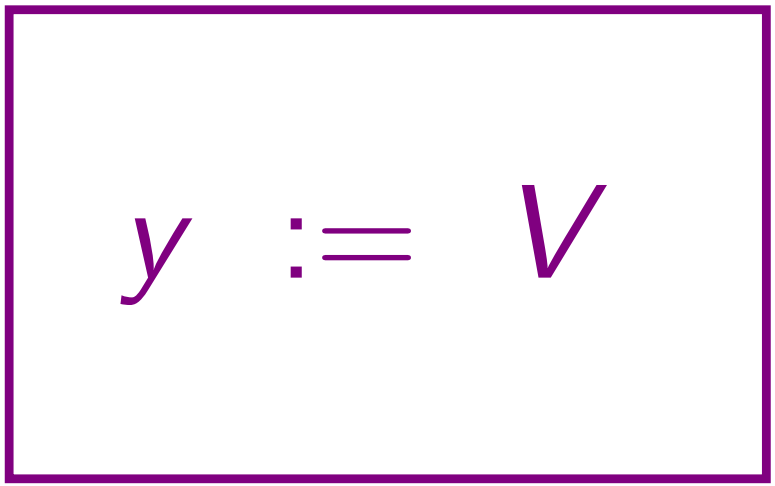
Calling a function



y: Address of argument V

q: Evaluate x at q

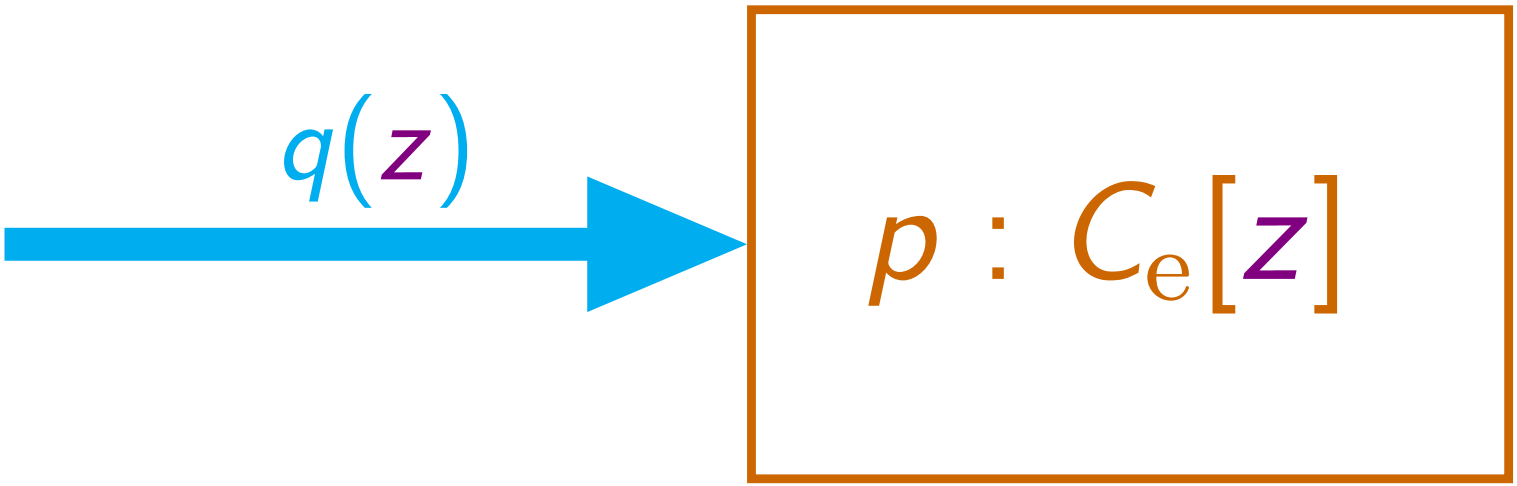
Calling a function



y: Address of argument V

q: Evaluate x at q

Calling a function



z: Address containing the evaluation of xV

q: Evaluate x at q

The Full Abstraction Problem

Full Abstraction for encoding call-by-value in the π-calculus

I. **Full Abstraction** for
   Milner's encoding of the call-by-value λ-calculus in the π-calculus

II. A new fixpoint theorem for weak behavioural equivalences

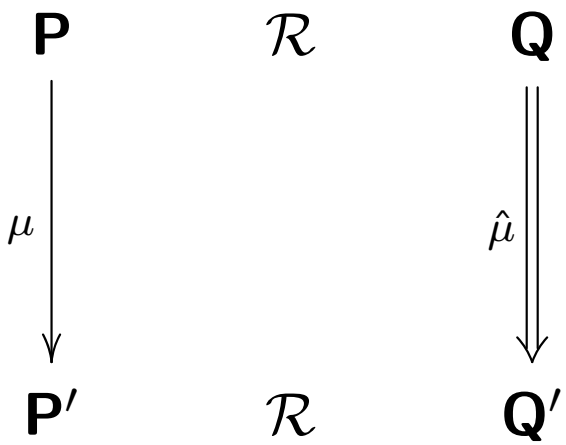
Full abstraction

Theorem

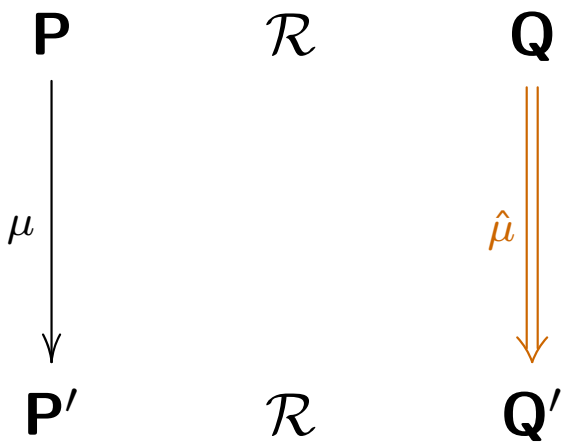
$$M \simeq_{e\eta} N \quad \textit{iff} \quad [\![M]\!] \approx [\![N]\!]$$

($\approx$ is bisimilarity (or contextual equivalence) in π)

Bisimulations (weak)



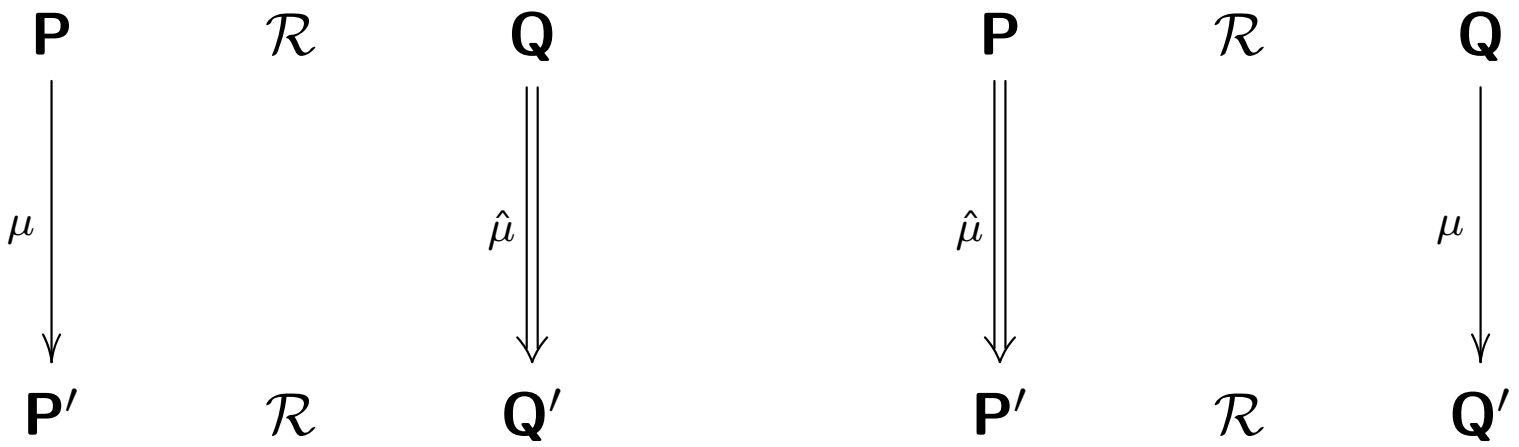
Bisimulations (weak)



$$\overset{\hat{\mu}}{\Rightarrow} := \left\{ \begin{array}{c} \overset{\tau}{\rightarrow}{}^{*} \overset{\mu}{\rightarrow} \overset{\tau}{\rightarrow}{}^{*} \quad \text{if } \mu \neq \tau \\ \overset{\tau}{\rightarrow}{}^{*} \quad \text{if } \mu = \tau \end{array} \right\}$$

weak transitions: τ is **invisible**

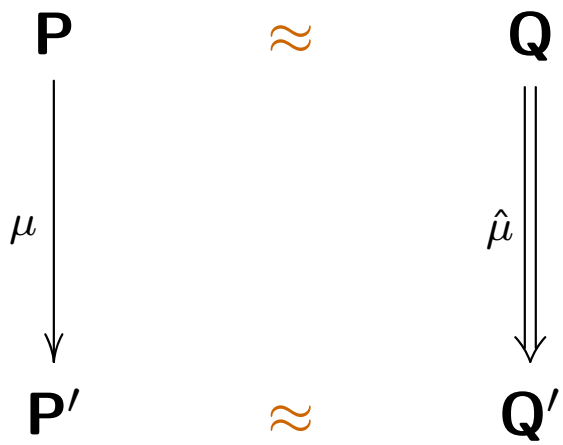
Bisimulations (weak)



$$\stackrel{\hat{\mu}}{\Rightarrow} := \left\{ \begin{array}{c} \stackrel{\tau}{\rightarrow}^* \stackrel{\mu}{\rightarrow} \stackrel{\tau}{\rightarrow}^* \quad \text{if } \mu \neq \tau \\ \stackrel{\tau}{\rightarrow}^* \quad \text{if } \mu = \tau \end{array} \right\}$$

weak transitions: τ is **invisible**

Bisimulations (weak)

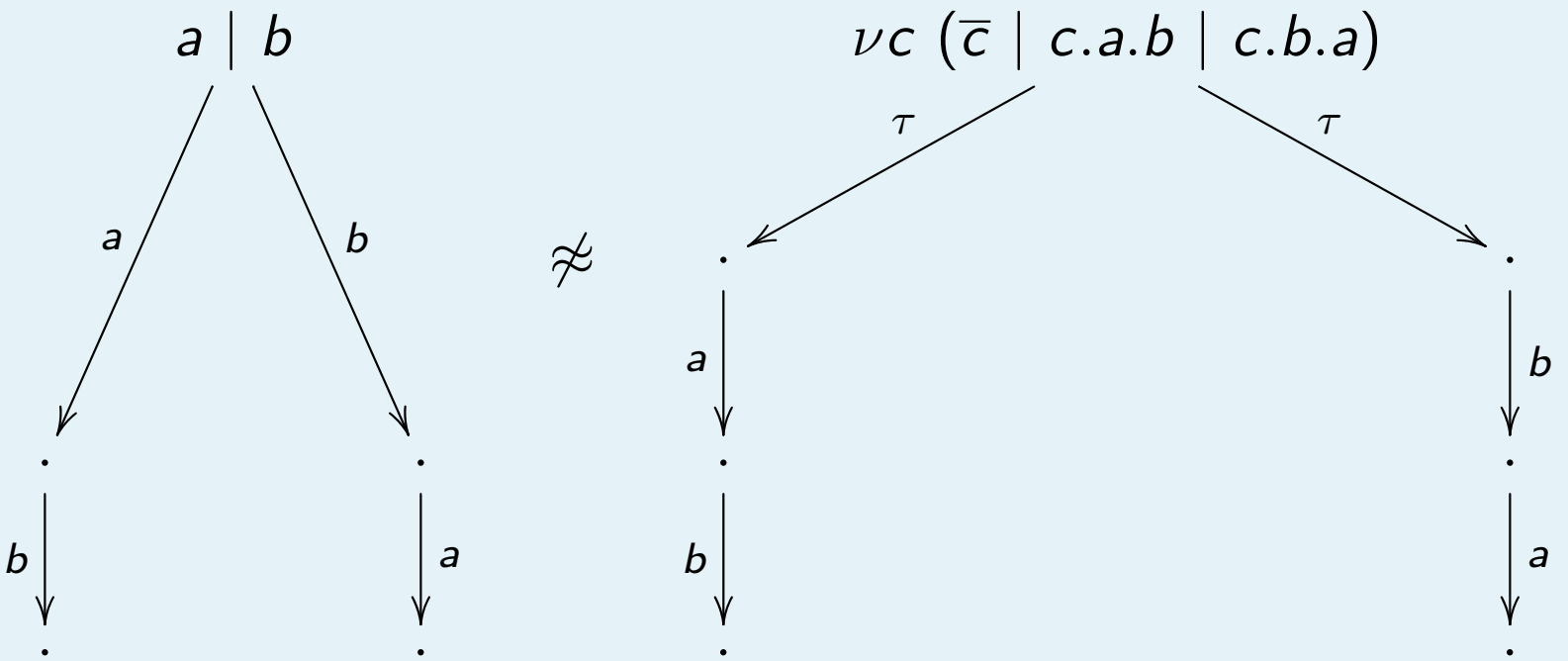


$$\approx \; = \; \cup \mathcal{R}$$

$\approx$: (weak) bisimilarity

Bisimulations (weak)

Bisimulations are matching trees



Full Abstraction for *call-by-name* and *call-by-value*

The **evaluation strategy** determines the induced equivalence

Two encodings, for two evaluation strategies:

The ***call-by-name*** λ-calculus

- Solved [Sangiorgi92]
- Fully abstract for **Lévy-Longo trees**:
  (standard equivalence for cbn)
- Proof relies on **up-to techniques**

Full Abstraction for *call-by-name* and *call-by-value*

The **evaluation strategy** determines the induced equivalence

Two encodings, for two evaluation strategies:

The ***call-by-name*** λ-calculus

- Solved [Sangiorgi92]
- Fully abstract for **Lévy-Longo trees**: (standard equivalence for cbn)
- Proof relies on **up-to techniques**
  $\rightarrow$ **sensitive to efficiency**

The ***call-by-value*** λ-calculus

- This work
- Equivalences not as well established or understood
- Introduces **inefficiencies**
  $\rightarrow$ **up-to techniques fail**

efficiency: number of reductions/synchronisations

Full abstraction for *call-by-value*

- open terms → we need to isolate the part being evaluated
- *call-by-name*: evaluation happens at top-level (main function)
- *call-by-value*: evaluation can happen deep in a term

→ a term has to be extracted from its context:

$$C_e[xV] \rightsquigarrow (\lambda z.C_e[z])(xV) \quad \text{or} \quad C[P] \rightsquigarrow C[\overline{z}] \mid !z.P$$

$\overline{z}$ **triggers** the replicated $!z.P$

- This transformation adds inefficiencies
- Up-to techniques are **sensitive to efficiency**:
  → we need to be always more efficient
  → hard-coded in the techniques

Full abstraction

Theorem

$$M \simeq_{e\eta} N \quad \textit{iff} \quad [\![M]\!] \simeq [\![N]\!]$$

($\simeq$ is barbed congruence or contextual equivalence in π)

- Completeness: $M \simeq_{e\eta} N \quad \Rightarrow \quad [\![M]\!] \simeq [\![N]\!]$
  - introduces inefficiencies:
    $$C_e[xV] \rightsquigarrow (\lambda z.C_e[z])(xV)$$
  - up-to techniques fail

The Proof Technique

Unique solution of equations

I. **Full Abstraction** for
Milner's encoding of the call-by-value λ-calculus in the π-calculus

II. A new **fixpoint theorem** for weak behavioural equivalences

The unique solution technique

$$\mathrm{X} = \mathbf{E}[\mathrm{X}]$$

(**E** is a **context**, that is, a **term with a hole**)

E has a **unique solution** for equivalence $=$ whenever:

If $\mathbf{P} = \mathbf{E}[\mathbf{P}]$ **and** $\mathbf{Q} = \mathbf{E}[\mathbf{Q}]$

Then $\mathbf{P} = \mathbf{Q}$

The completeness proof

Lemma (Completeness)

$$M \simeq_{e\eta} N \quad \textit{implies} \quad [\![M]\!] \approx [\![N]\!]$$

1. Build a system of equations (from bisimulation $\mathcal{R}$ s.t $M \mathcal{R} N$)
2. Prove $[\![M]\!]$ and $[\![N]\!]$ are solutions
3. Prove the system has a unique solution

The equations

If $M \simeq_{e\eta} N$, they have the same tree, given by a bisimulation $\mathcal{R}$

- One equation per node (infinite system of equations)

$$X_{M,N} = \mathbf{E}[X_{M_1,N_1}, \ldots, X_{M_i,N_i}]$$

$\rightarrow$ The type of node determines the equation

The Equations

1. If M, N diverge, the equation is

$$X_{M,N} = \mathbf{0}$$

The Equations

1. If M, N diverge, the equation is

$$X_{M,N} = \mathbf{0}$$

2. If $M \Rightarrow x$, $N \Rightarrow x$

$$X_{M,N} = [\![x]\!]$$

Th

The Equations

1. If M, N diverge, the equation is

$$X_{M,N} = \mathbf{0}$$

2. If $M \Rightarrow x$, $N \Rightarrow x$

$$X_{M,N} = [\![x]\!]$$

3. If $M \Rightarrow \lambda x.M'$, $N \Rightarrow \lambda x.N'$

$$X_{M,N} = [\![\lambda x.X_{M',N'}]\!]$$

The Equations

1. If M, N diverge, the equation is

$$X_{M,N} = \mathbf{0}$$

2. If $M \Rightarrow x$, $N \Rightarrow x$

$$X_{M,N} = [\![x]\!]$$

3. If $M \Rightarrow \lambda x.M'$, $N \Rightarrow \lambda x.N'$

$$X_{M,N} = [\![\lambda x.X_{M',N'}]\!]$$

4. If $M \Rightarrow C_e[xV]$ and $N \Rightarrow C'_e[xV']$

$$X_{M,N} = [\![(\lambda z.X_{C_e[z],C'_e[z]})(xX_{V,V'})]\!] \text{ (optimised)}$$

The equations

If $M \simeq_{e\eta} N$, they have the same tree, given by a bisimulation $\mathcal{R}$

- One equation per node (infinite system of equations)

$$X_{M,N} = \mathbf{E}[X_{M_1,N_1}, \ldots, X_{M_i,N_i}]$$

$\rightarrow$ The type of node determines the equation

- The structure of the proof follows the structure of the tree
- Equations represent **normal forms** in the π-calculus

The completeness proof

Lemma (Completeness)

$$M \simeq_{e\eta} N \quad \textit{implies} \quad [\![M]\!] \approx [\![N]\!]$$

1. Build a system of equations (from bisimulation $\mathcal{R}$ s.t $M \mathcal{R} N$)
2. Prove $[\![M]\!]$ and $[\![N]\!]$ are solutions
3. Prove the system has a unique solution

Unique solution of equations

From CSP to CCS

I. Full Abstraction

II. A new fixpoint theorem for weak behavioural equivalences
- Builds on existing theory of unique solution of equations
  Hoare[1978], Milner [1980], Roscoe [1982], Sangiorgi [2015]...
- Two important schools in concurrency: **CSP** and **CCS**
- Adapted from Roscoe's unique solution Theorem in CSP [1988]
- Operational, generalizable theorem

Equations and unique solution

$$X = \mathbf{E}[X]$$

E is a **context**

π-calculus, CCS, CSP, λ-calculus, ...

E has a **unique solution** for $\approx$ whenever:

If $\mathbf{P} \approx \mathbf{E}[\mathbf{P}]$ **and** $\mathbf{Q} \approx \mathbf{E}[\mathbf{Q}]$

Then $\mathbf{P} \approx \mathbf{Q}$

Equations and unique solution

$$X = \mathbf{E}[X]$$

E is a **context**

Example

$$X = a.\tau.X$$

All solutions are $\approx a.a.a.a.a\ldots$

Equations always have solutions

$$\mathrm{X} = \mathbf{E}[\mathrm{X}]$$

Has at least one a solution:

- Consider the constant $\mathbf{K_E} := \mathbf{E}[\mathbf{K_E}]$
- $\mathbf{K_E} = \underbrace{\mathbf{E}[\mathbf{E}[\mathbf{E}[\mathbf{E}[\ldots]]]]}_{\mathbf{E}^\infty\text{: syntactic solution}}$ is always solution of $\mathrm{X} = \mathbf{E}[\mathrm{X}]$
- It has the **least possible behaviours of all solutions**
- It is the **minimal** solution
  (least fixed point)

Equations do not always have a unique solution

$\mathrm{X} = \mathrm{X}$, $\mathrm{X} = \tau.\mathrm{X}$ do not have a unique solution:

$$\forall \mathbf{P}, \mathbf{P} \approx \tau.\mathbf{P}$$

Equations do not always have a unique solution

$\mathrm{X} = \mathrm{X}$, $\mathrm{X} = \tau.\mathrm{X}$ do not have a unique solution:

$$\forall \mathbf{P}, \mathbf{P} \approx \tau.\mathbf{P}$$

1. **No prefix** to constrain behavior
2. **Weak prefix (τ)** also does not constrain behavior

$$\mathrm{X} = \mathrm{a}.\overline{\mathrm{b}}.\mathrm{X}$$

Unique solution up to $\approx$: $\mathrm{a}.\overline{\mathrm{b}}.\mathrm{a}.\overline{\mathrm{b}}.\mathrm{a}\ldots$
Is this enough?

Failure of unique solution

$$X \;=\; b.\,(\,\overline{b} \mid X\,)$$

If $b.\,\mathbf{P}$ is a solution, then $b.\,(\,\mathbf{P} \mid \mathbf{Q}\,)$ is a solution (for any $\mathbf{Q}$)

Hence, many $\not\approx$ **solutions**

$\rightarrow$ prefixes can be **erased**

CSP

$\mathtt{a} \backslash \mathtt{a} \xrightarrow{\tau}$

CCS

$\mathtt{a} \mid \overline{\mathtt{a}} \xrightarrow{\tau}$

Unique Solution theorems

Unique solution Theorems in CSP:
(Hoare [1978] & Roscoe [1982])

- Remove hiding operator $\backslash$
- Restrict hiding operator $\backslash$

↓

Roscoe's Theorem [1988]:

- Allow $\backslash$, forbid divergences

Unique solution Theorems in CCS:
(Milner [1980])

- Remove parallel composition $|$
- Or consider strong bisimulation

↓

Sangiorgi's contractions [2015]:

- Enforce efficiency of solutions

Unique Solution with no divergences for CCS

Divergences and unique solution

$$X = \tau.\ X$$

- $\mathbf{E}^{\infty} = \tau.\tau.\tau.\tau \ldots$

Divergences and unique solution

$$X = \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid X\ )$$

- $\forall \mathbf{P},\ \mathtt{b}.\ (\ \mathbf{P} \mid \overline{\mathtt{b}} \mid (\mathtt{b}.\overline{\mathtt{b}})^{\omega})$ **is solution**
- $\mathbf{E}^{\infty} = \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b} \ldots\ )\ )\ ) \xrightarrow{\mathtt{b}} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \ldots$

Divergences and unique solution

$$X = \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid X\ )$$

- $\forall \mathbf{P},\ \mathtt{b}.\ (\ \mathbf{P} \mid \overline{\mathtt{b}} \mid (\mathtt{b}.\overline{\mathtt{b}})^{\omega})$ **is solution**
- $\mathbf{E}^{\infty} = \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b} \ldots\ )\ )\ ) \xrightarrow{\mathtt{b}} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \ldots$
- **Divergence:** infinite sequence of $\xrightarrow{\tau}$ transitions

Divergences and unique solution

$$X = \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid X\ )$$

- $\forall \mathbf{P},\ \mathtt{b}.\ (\ \mathbf{P} \mid \overline{\mathtt{b}} \mid (\mathtt{b}.\overline{\mathtt{b}})^{\omega}\ )$ **is solution**
- $\mathbf{E}^{\infty} = \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b}.\ (\ \overline{\mathtt{b}} \mid \mathtt{b} \ldots\ )\ )\ ) \xrightarrow{\mathtt{b}} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \ldots$
- **Divergence:** infinite sequence of $\xrightarrow{\tau}$ transitions
- In CSP, the trace $\mathtt{bbbbb}\ldots \setminus \mathtt{b}$ is a a divergence
  $\rightarrow$ Roscoe's Unique Solution Theorem [1988]

Unique Solution for **divergence-free** equations

Theorem (Unique Solution for divergence free equations)

If E^∞ has no divergences, then guarded equation $X = E[X]$ has a unique solution for $\approx$.

- P has a divergence: $P \xrightarrow{\mu_1} \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} \xrightarrow{\tau} \xrightarrow{\tau} \ldots \xrightarrow{\tau} \ldots$
- **Syntactic solution** $E^\infty := E[E[E[\ldots]]]$

Unique Solution for **divergence-free** equations

Theorem (Unique Solution for divergence free equations)

If E^∞ has no divergences, then guarded equation $X = E[X]$ has a unique solution for $\approx$.

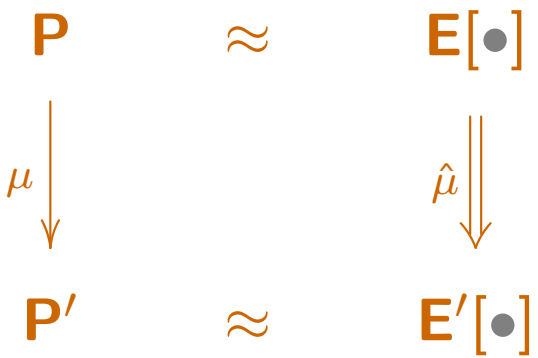
- P has a divergence: $P \xrightarrow{\mu_1} \xrightarrow{\mu_2} \dots \xrightarrow{\mu_n} \xrightarrow{\tau} \xrightarrow{\tau} \dots \xrightarrow{\tau} \dots$
- **Syntactic solution** $E^\infty := E[E[E[\dots]]]$

$\rightarrow$ **Divergences**: Efficiency only becomes important **at the limit**

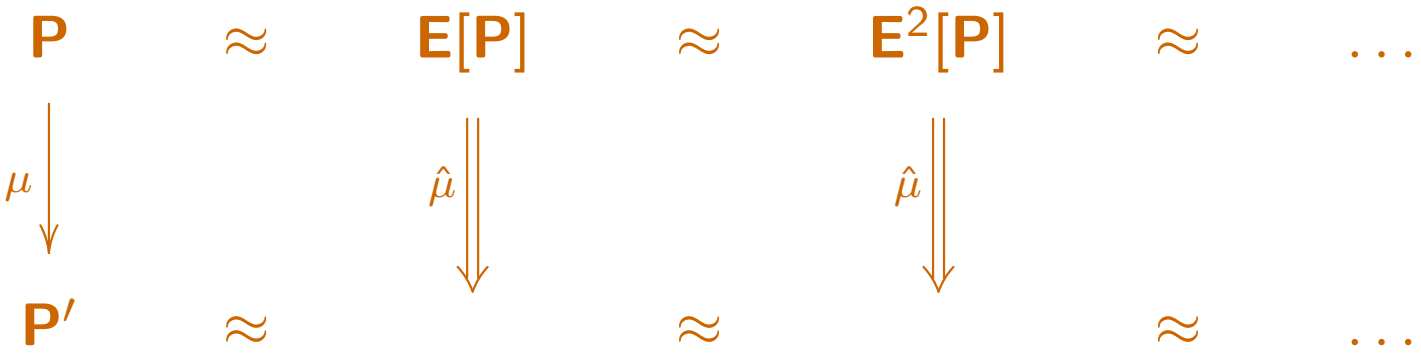
$\rightarrow$ this guarantee is insensitive to **local** inefficiency

Ensuring unique solution

Ensuring unique solution



Ensuring unique solution



Ensuring unique solution

$$
\begin{array}{ccccccccc}
\mathbf{P} & \approx & \mathbf{E}[\mathbf{P}] & \approx & \mathbf{E}^2[\mathbf{P}] & \approx & \ldots & \approx & \mathbf{E}^n[\mathbf{P}] \\
\Big\downarrow{\scriptstyle \mu} & & \Big\Downarrow{\scriptstyle \hat{\mu}} & & \Big\Downarrow{\scriptstyle \hat{\mu}} & & & & \Big\Downarrow{\scriptstyle \hat{\mu}} \\
\mathbf{P}' & \approx & & \approx & & \approx & \ldots & \approx & \mathbf{E}'[\mathbf{P}]
\end{array}
$$

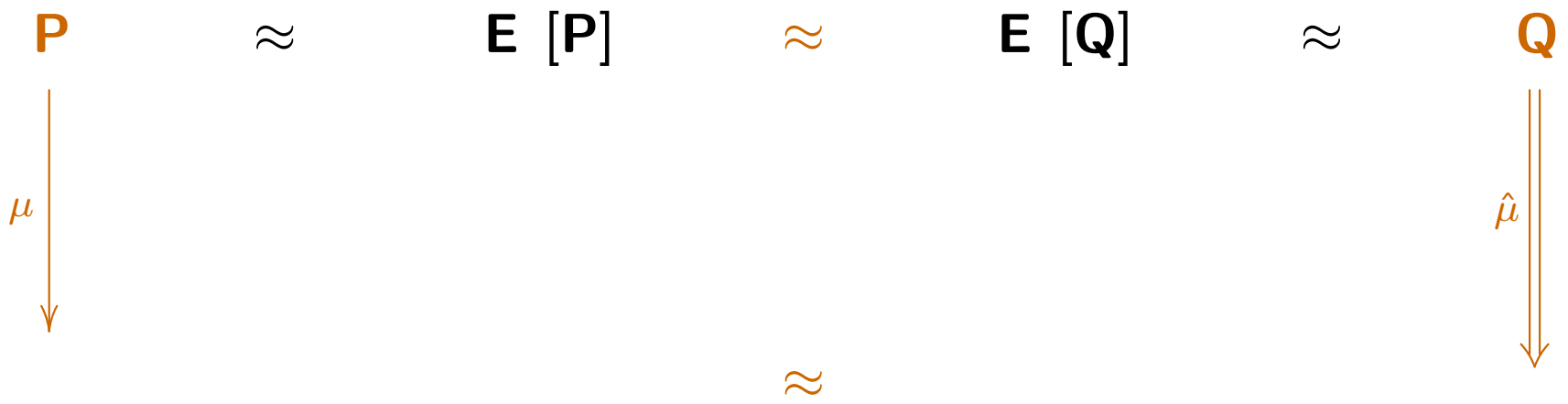
Unique solution and context transitions

$$\mathbf{P} \quad \approx \quad \mathbf{E}\ [\mathbf{P}] \qquad\qquad \mathbf{E}\ [\mathbf{Q}] \quad \approx \quad \mathbf{Q}$$

$\mathbf{P}$ are $\mathbf{Q}$ solutions of $\mathrm{X} = \mathbf{E}[\mathrm{X}]$.
We show unique solution:

$$\mathbf{P} \approx \mathbf{Q}$$

Unique solution and context transitions



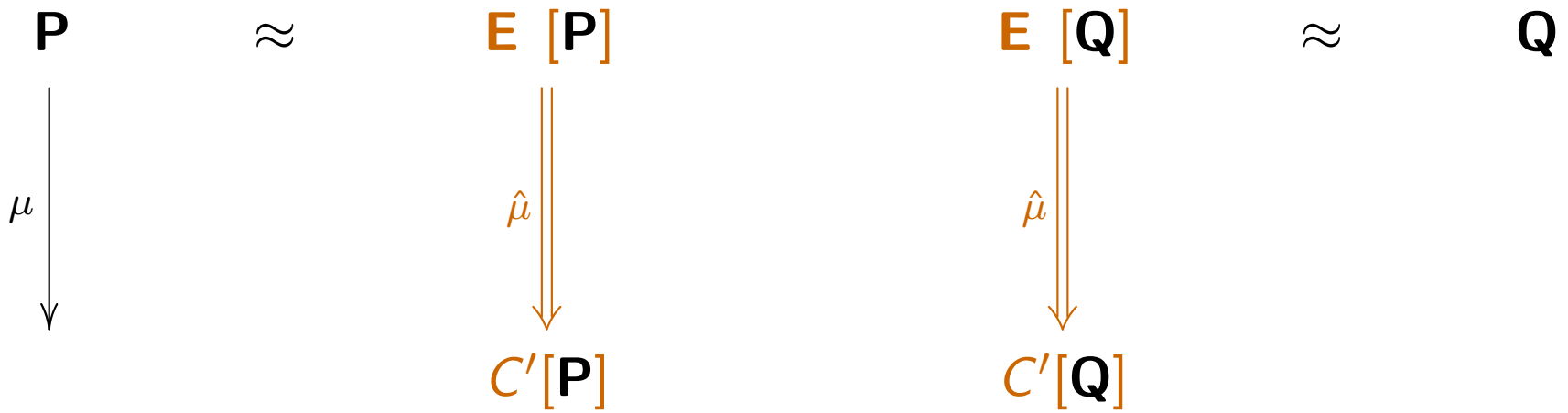
Challenges of **P**

Unique solution and context transitions

Unique solution and context transitions

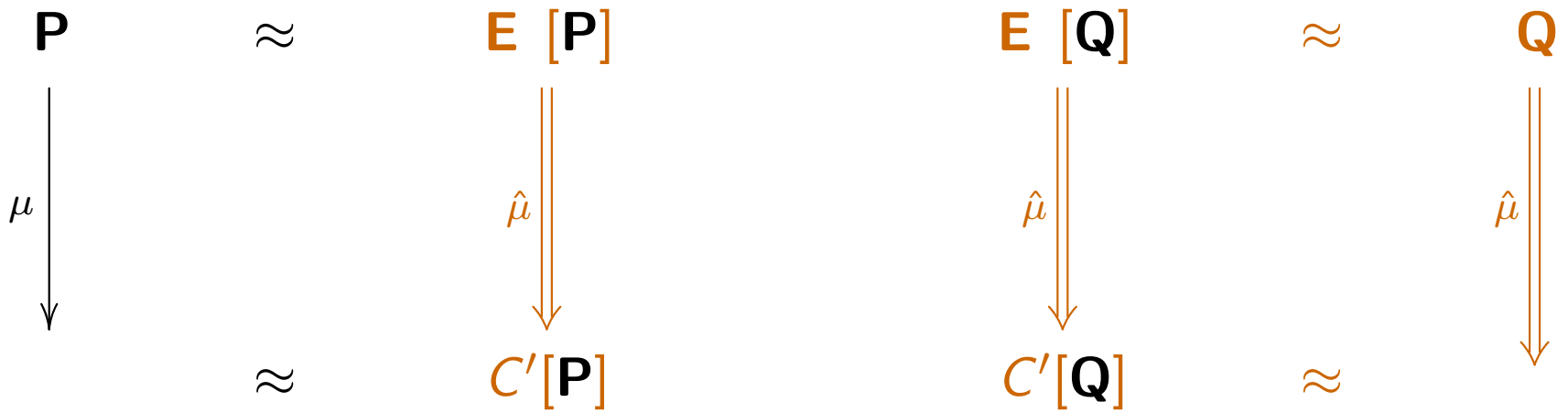
Assume **this transition** is a **context transition** of **E**

Unique solution and context transitions



Assume **this transition** is a **context transition** of **E**
E[**Q**] can match it;

Unique solution and context transitions



Assume **this transition** is a **context transition** of **E**

E[**Q**] can match it;

Q too.

Unique solution and context transitions

Context transitions

- Transitions occuring **independently of the process inside:**

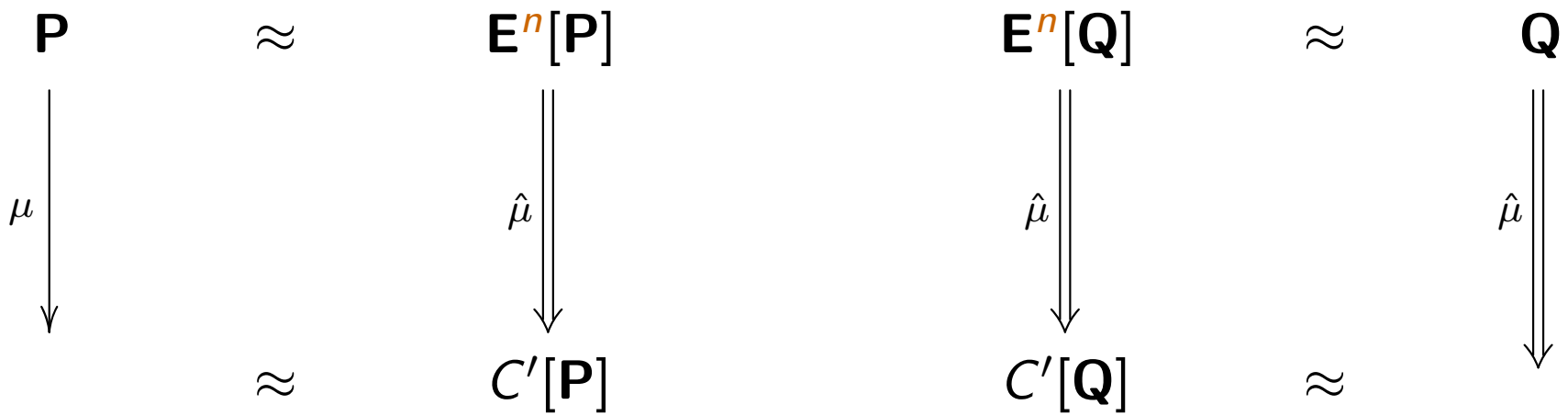
$$C \xrightarrow{\mu} C' \Leftrightarrow \forall P, C[P] \xrightarrow{\mu} C'[P]$$

- Generalizable concept
- Related to weak guardedness
  (guarded context $\Rightarrow$ context transition)

$$\tau.P \xrightarrow{\tau} \text{ is a context transition}$$

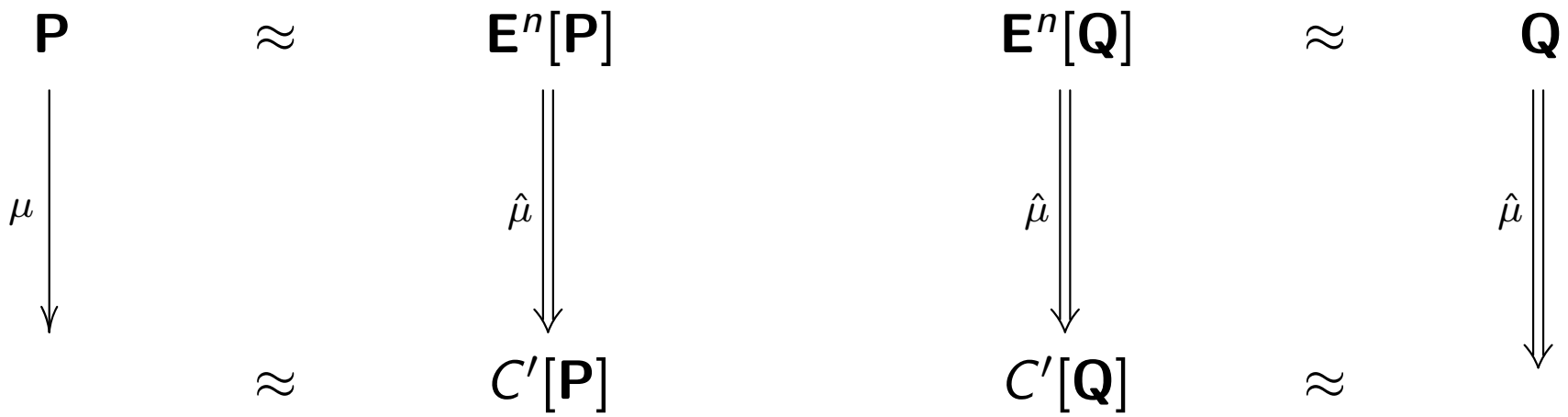
The proof needs to be written using a **strong** semantics.

Unique solution and context transitions



P is also solution of $\mathrm{X} = \mathbf{E}^n[\mathrm{X}]$

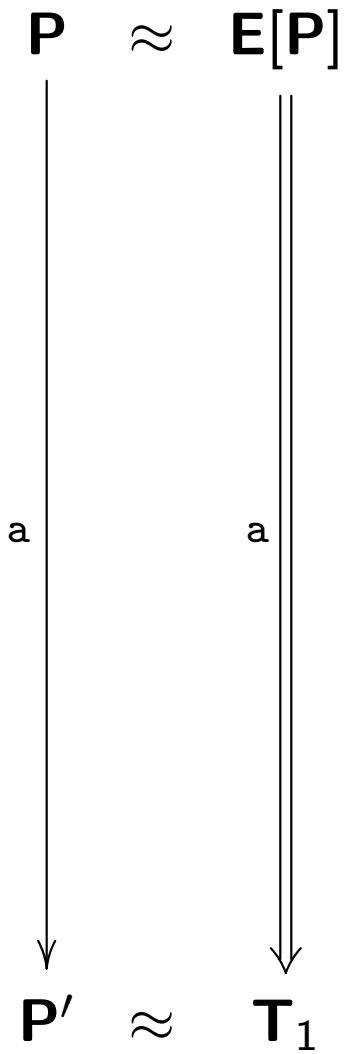
Unique solution and context transitions



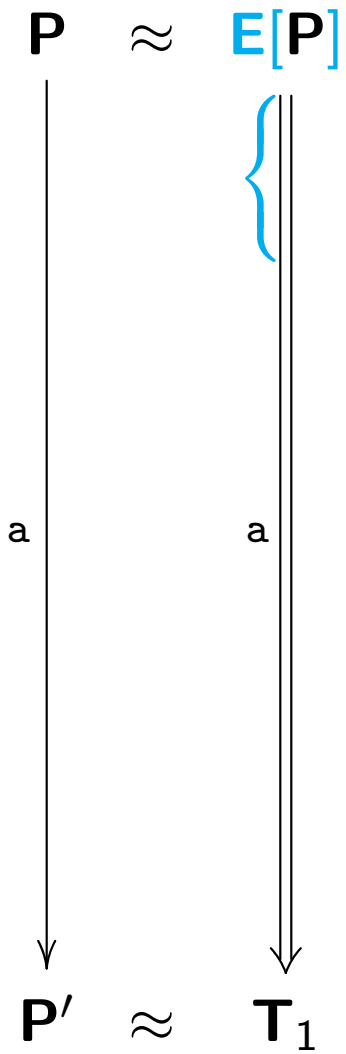
If any transition of **P** can be matched by a transition of $\mathbf{E}^n$

→ **Unique solution**

Constructing context transitions

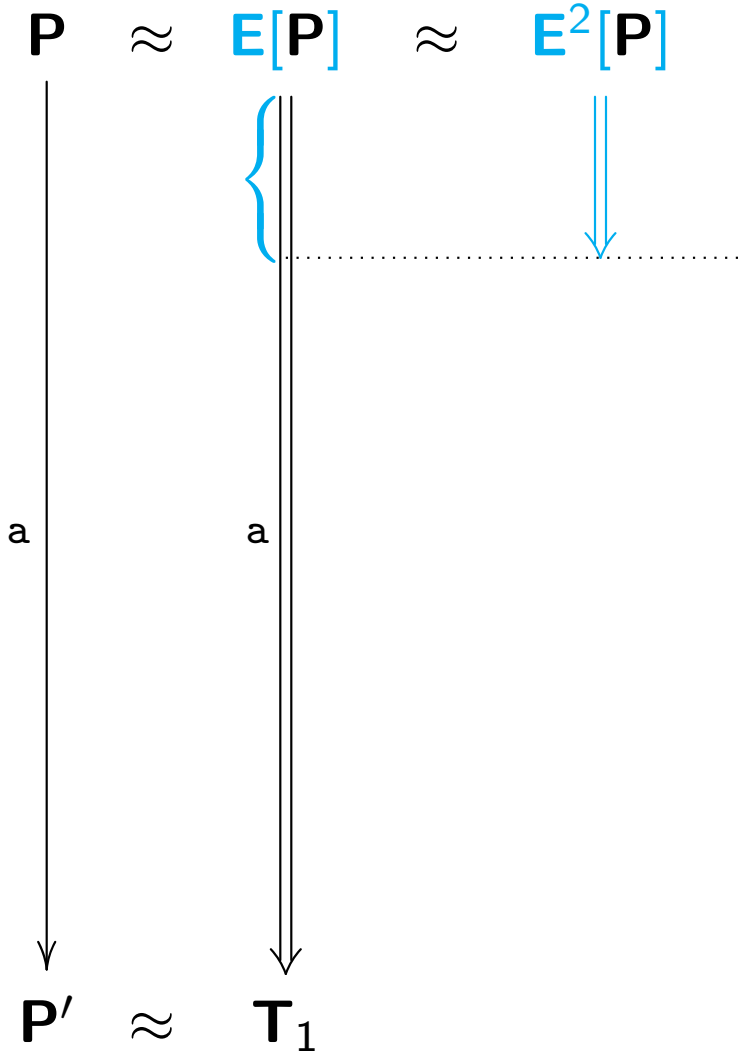


Constructing context transitions

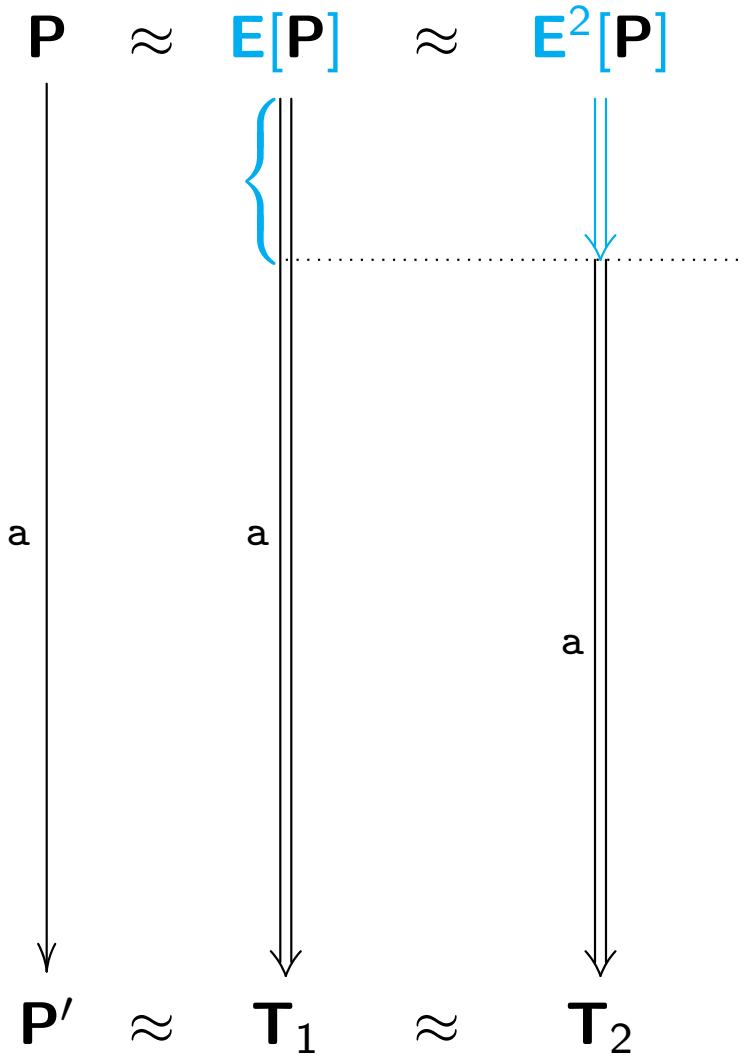


E is guarded

Constructing context transitions

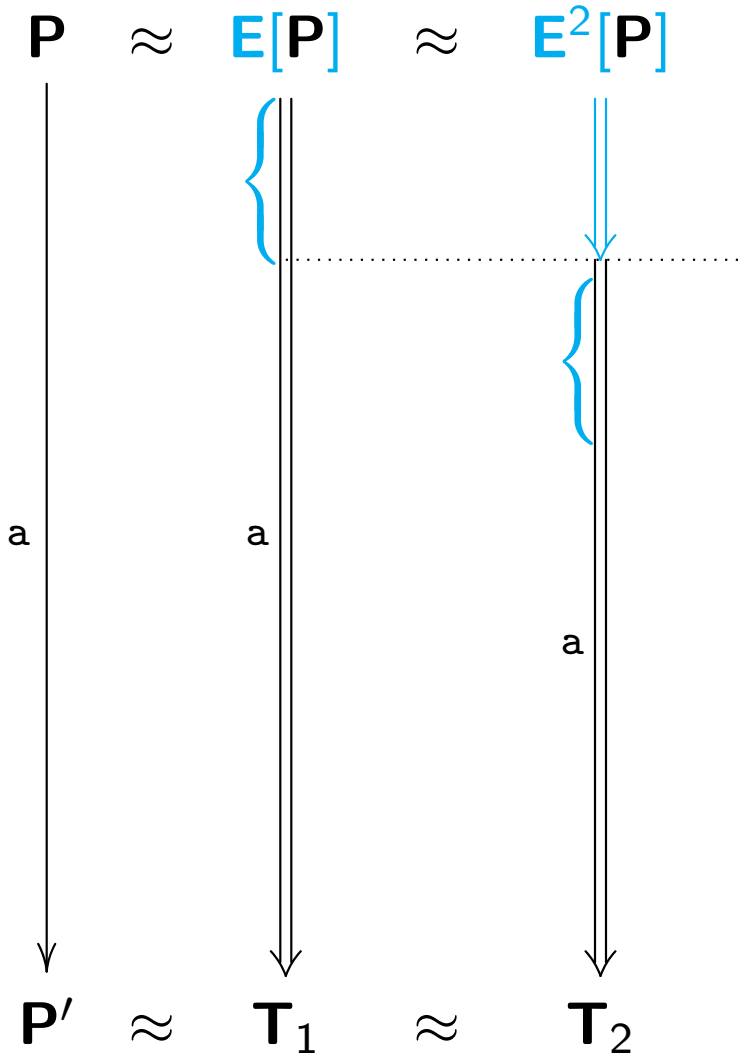


Constructing context transitions



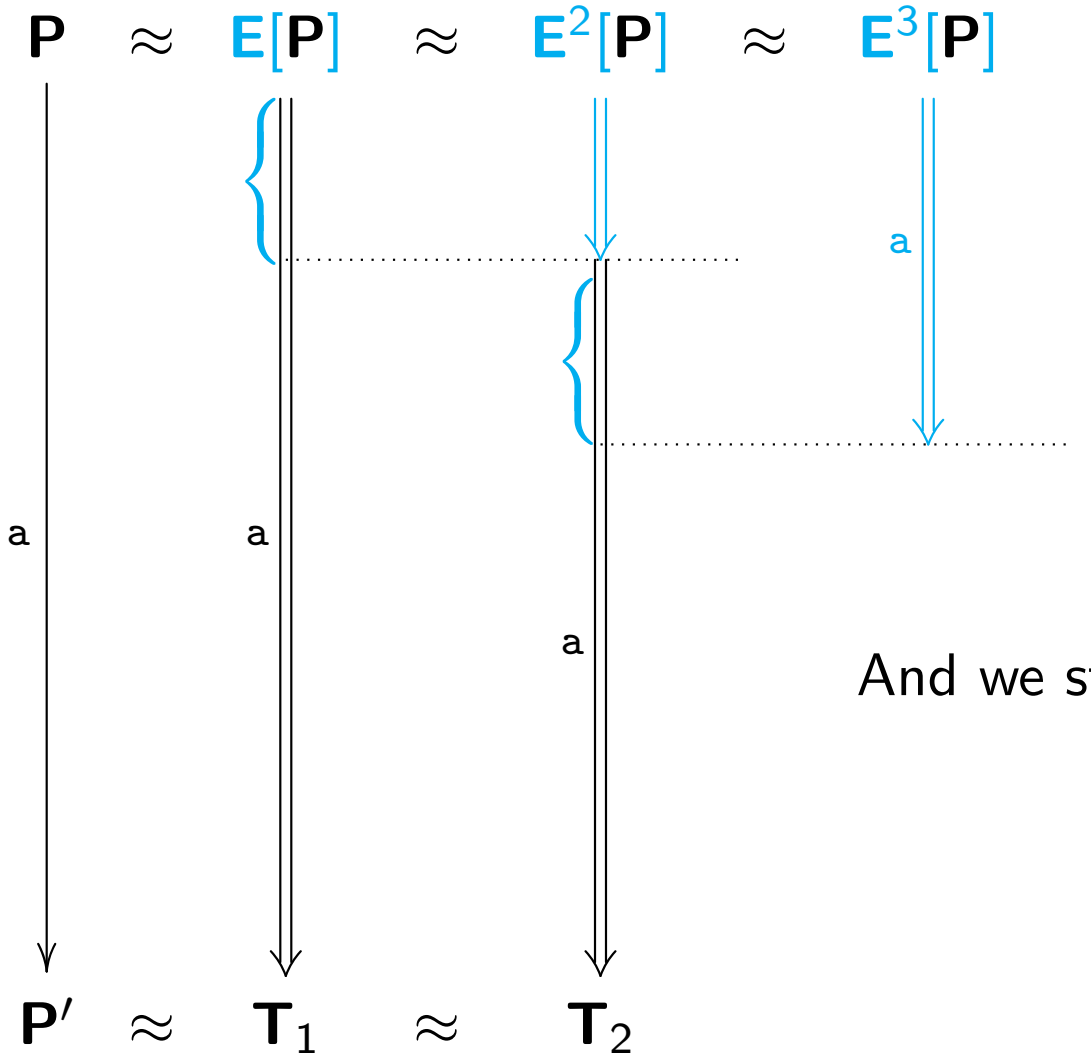
We complete by bisimilarity

Constructing context transitions



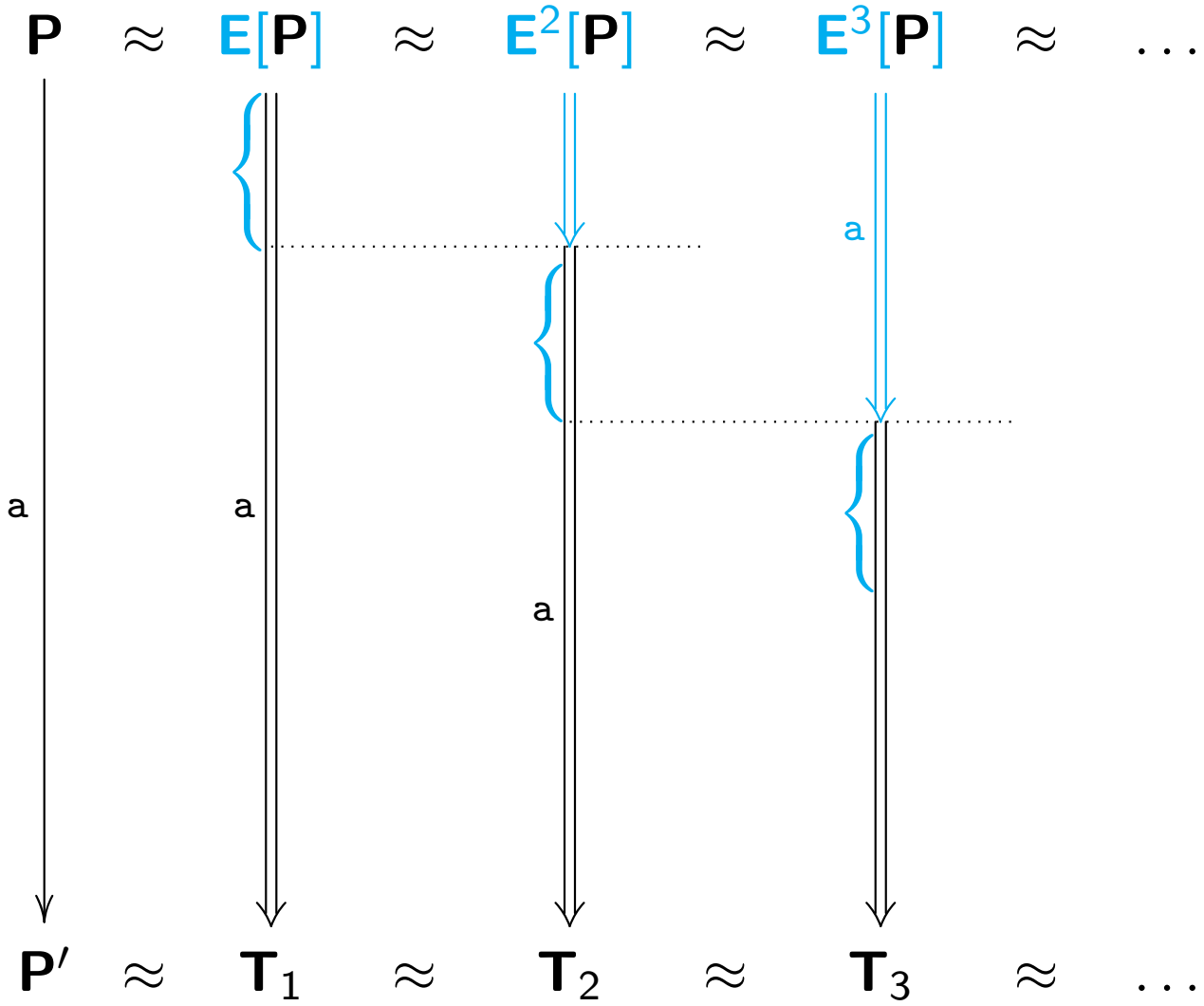
And we start again. . .

Constructing context transitions

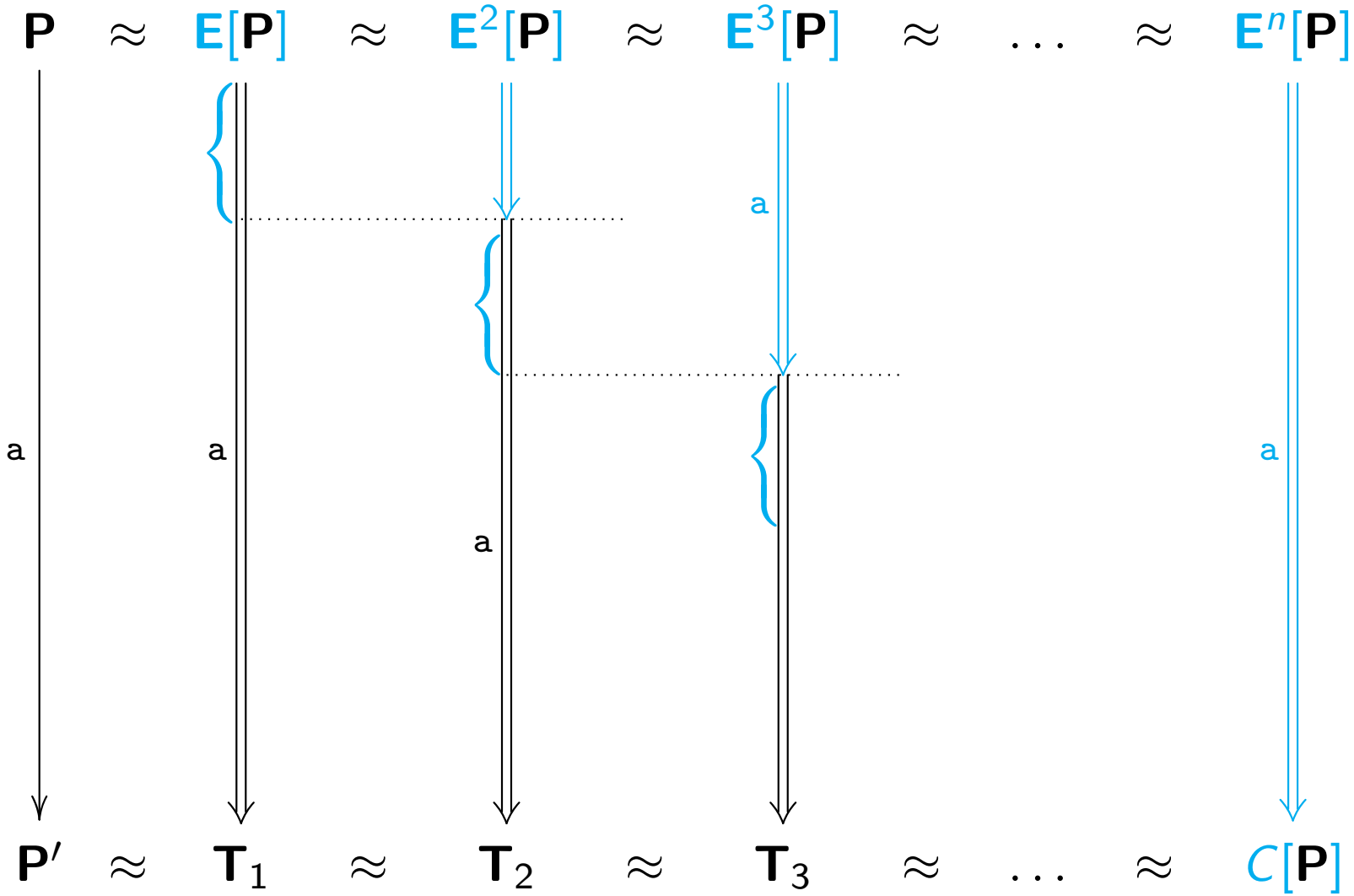


And we start again. . .

Constructing context transitions



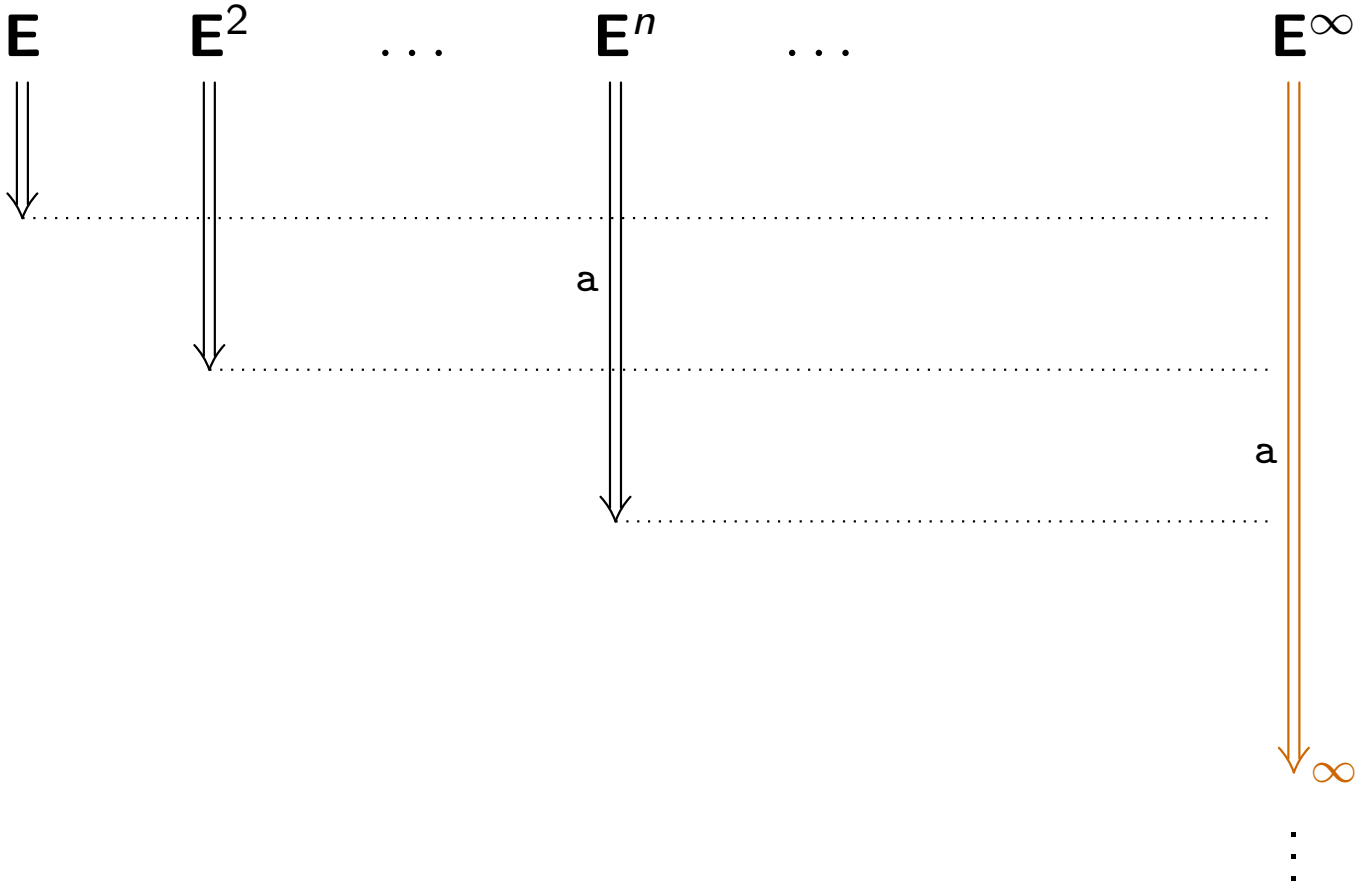
Constructing context transitions



Limit

If the construction never stops:

$$\Rightarrow \textbf{divergence of } \mathbf{E}^\infty \ (= \mathbf{E}[\mathbf{E}[\mathbf{E}[\ldots]]])$$



Divergences built this way are **non-innocuous**

Other contributions to the theory of Unique Solution

- **Fined-grained** analysis of **divergences**
  - Innocuous divergences, completeness results
- **Abstract formulation**, we only need a few **ingredients**:
  - **Context transitions**
  - **Guarded contexts** (mandatory context transitions)
  - Contexts are **congruences**
    $$\mathbf{P} \approx \mathbf{Q} \quad \Rightarrow \quad C[\mathbf{P}] \approx C[\mathbf{Q}]$$
  - Contexts **compose**
- Unique Solution for name-passing and **Higher-Order**
  - π-calculus
  - Higher-Order π-calculus
- traces, formats

The completeness proof

Lemma (Completeness)

$$M \simeq_{e\eta} N \quad \textit{implies} \quad [\![M]\!] \approx [\![N]\!]$$

1. Build a system of equations (from bisimulation $\mathcal{R}$ s.t $M \mathcal{R} N$)
2. Prove $[\![M]\!]$ and $[\![N]\!]$ are solutions
3. Prove the system has a unique solution

Conclusion

Future works and questions

Conclusion & Future Works

- **Guardedness** and **non-divergence** guarantee **a unique solution**
  - Bridge between **CCS** and **CSP**
  - Generalizable proof (languages, equivalences, abstract formulation)
  - Fined-grained analysis of divergences
- Comparison with **up-to techniques**:
  - widely studied bisimulation enhancements
  - Some proofs are not possible with up-to techniques
  - 'Up to context techniques and unique solution are (often) the same' [Sangiorgi '15]
  - Correspondence with Pous' [08] up-to $\approx$ with termination?
  - Correspondence **fails for higher-order** (π)

Conclusion & Future work: π-calculus encodings

- **Full abstraction** between Milner's π-calculus encoding and Lassen's eager normal form bisimilarity
- Full abstraction for variants of the encoding
  - **Parallel *call-by-value***: distinguishes diverging terms
  - Full abstraction for **contextual equivalence** ?

Co

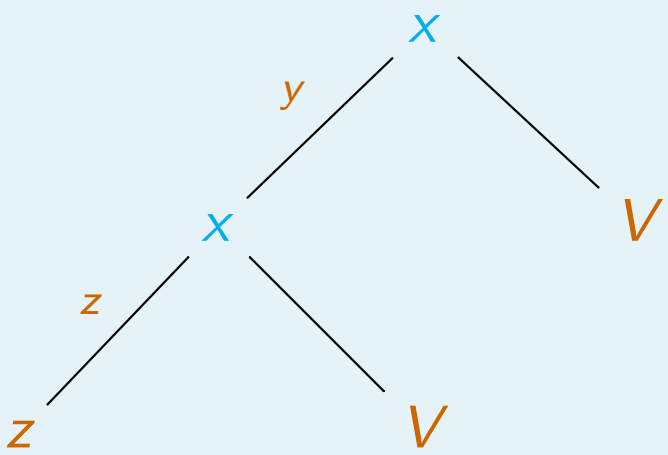
Contextual equivalence vs. eager normal form bisimilarity

[Lassen07]: Eager normal form is fully abstract for $\lambda\mu$+references

Counter-examples:

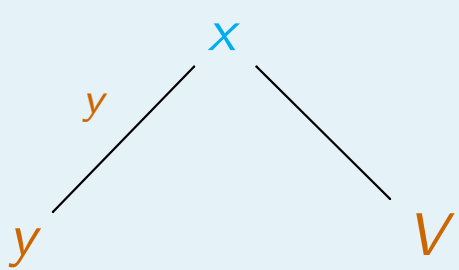
1. $(\lambda y.\Omega)(x\,V) \not\approx_{e\eta} \Omega$
   - $\rightarrow$ Distinguishable by raising an exception in x
2. $(\lambda y.x\,V)(x\,V) \not\approx_{e\eta} x\,V$
   - $\rightarrow$ Distinguishable by using a reference in x

$(\lambda y.x\,V)(x\,V)$



$\not\approx_e$

$x\,V$



Conclusion & Future work: π-calculus encodings

- **Full abstraction** between Milner's π-calculus encoding and Lassen's eager normal form bisimilarity
- Full abstraction for variants of the encoding
  - **Parallel *call-by-value***: distinguishes diverging terms
  - Full abstraction for **contextual equivalence** ?
- Links with **game semantics**
  - Connections between games and π-calculus [Hyland,Ong 95] [Honda, Laurent 11]
  - Session types [Berger,Honda,Yoshida 01] [Toninho,Yoshida 18]

Questions

Thank you for your attention!