Session-Typed Concurrent Programming
Lecture 2

Stephanie Balzer
Carnegie Mellon University

OPLSS 2021
June 24, 2021

Recap

Recap

- Roadmap and learning objectives

Recap

- Roadmap and learning objectives
- Message-passing concurrent programming
  - pi-calculus as formal model
  - nondeterminism

Recap

- Roadmap and learning objectives
- Message-passing concurrent programming
  - pi-calculus as formal model
  - nondeterminism
- Session types as types of message-passing concurrency
  - challenge: preservation because type changes with protocol
  - strategies: (a) disallow aliasing or (b) control aliasing

Recap

- Roadmap and learning objectives
- Message-passing concurrent programming
  - pi-calculus as formal model
  - nondeterminism
- Session types as types of message-passing concurrency
  - challenge: preservation because type changes with protocol
  - strategies: (a) disallow aliasing or (b) control aliasing
- Intuitionistic linear logic as a foundation for session types

Recap

- Roadmap and learning objectives
- Message-passing concurrent programming
  - pi-calculus as formal model
  - nondeterminism
- Session types as types of message-passing concurrency
  - challenge: preservation because type changes with protocol
  - strategies: (a) disallow aliasing or (b) control aliasing
- Intuitionistic linear logic as a foundation for session types

we’ll resume here

Intuitionistic linear logic session types

Intuitionistic linear logic session types

Types:

A, B	$\triangleq$	$A \otimes B$	multiplicative conjunction	"channel output"
		$A \multimap B$	multiplicative implication	"channel input"
		$A \,\&\, B$	additive conjunction	"external choice"
		$A \oplus B$	additive disjunction	"internal choice"
		$\mathbf{1}$	unit for $\otimes$	"termination"

Queue session type:

$$\begin{aligned} \mathsf{queue}\ A = \&\{&\mathsf{enq} : A \multimap \mathsf{queue}\ A, \\ &\mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{queue}\ A\}\} \end{aligned}$$

Typing judgment and rules

Intuitionistic linear sequent:

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)$$

"Process P offers a session of type A along channel x using session A_1, …, A_n provided along channels x_1, …, x_n."

Inference rule:

premise / conclusion

$$\frac{\Delta' \vdash Q :: (x : A')}{\Delta \vdash P; Q :: (x : A)}$$

↑ bottom-up reading

Left and right rules:

$$\frac{\Delta', x : B \vdash Q :: (z : C)}{\Delta, x : A \diamond B \vdash P; Q :: (z : C)} \diamond_L \qquad \frac{\Delta' \vdash Q :: (x : B)}{\Delta \vdash P; Q :: (x : A \diamond B)} \diamond_R$$

Connectives so far

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \textsf{send}\ x\ y; P :: (x : A \otimes B)} \otimes_R \qquad \frac{\Delta, x : B, y : A \vdash Q_y :: (z : C)}{\Delta, x : A \otimes B \vdash y \leftarrow \textsf{recv}\ x; Q_y :: (z : C)} \otimes_L$$

$$\frac{\Delta, y : A \vdash P_y :: (x : B)}{\Delta \vdash y \leftarrow \textsf{recv}\ x; P_y :: (x : A \multimap B)} \multimap_R \qquad \frac{\Delta, x : B \vdash Q :: (z : C)}{\Delta, x : A \multimap B, y : A \vdash \textsf{send}\ x\ y; Q :: (z : C)} \multimap_L$$

$$\frac{\Delta \vdash P :: (x : A)}{\Delta \vdash x.\textsf{inl}; P :: (x : A \oplus B)} \oplus_{R_1} \qquad \frac{\Delta \vdash P :: (x : B)}{\Delta \vdash x.\textsf{inr}; P :: (x : A \oplus B)} \oplus_{R_2}$$

$$\frac{\Delta, x : A \vdash Q_1 :: (z : C) \qquad \Delta, x : B \vdash Q_2 :: (z : C)}{\Delta, x : A \oplus B \vdash \textsf{case}\ x\ \textsf{of}(Q_1, Q_2) :: (z : C)} \oplus_L$$

$$\frac{\Delta \vdash P_1 :: (x : A) \qquad \Delta \vdash P_2 :: (x : B)}{\Delta \vdash \textsf{case}\ x\ \textsf{of}(P_1, P_2) :: (x : A \,\&\, B)} \&_R$$

$$\frac{\Delta, x : A \vdash Q :: (z : C)}{\Delta, x : A \,\&\, B \vdash x.\textsf{inl}; Q :: (z : C)} \&_{L_1} \qquad \frac{\Delta, x : B \vdash Q :: (z : C)}{\Delta, x : A \,\&\, B \vdash x.\textsf{inr}; Q :: (z : C)} \&_{L_2}$$

Unit for multiplicative conjunction - termination

Unit for multiplicative conjunction - termination

$$\dfrac{}{\vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

Unit for multiplicative conjunction - termination

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

Unit for multiplicative conjunction - termination

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

no orphan providers

Unit for multiplicative conjunction - termination

$$\dfrac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

Unit for multiplicative conjunction - termination

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

$$\frac{\vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \mathsf{wait}\ x; Q :: (z : C)}\ \mathbf{1}_L$$

Unit for multiplicative conjunction - termination

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

$$\frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \mathsf{wait}\ x; Q :: (z : C)}\ \mathbf{1}_L$$

Unit for multiplicative conjunction - termination

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

we have lost x!

$$\frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \textsf{wait}\ x; Q :: (z : C)}\ \mathbf{1}_L$$

Unit for multiplicative conjunction - termination

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

we have lost x!

$$\frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \textsf{wait}\ x; Q :: (z : C)}\ \mathbf{1}_L$$

→ no unit for & and $\oplus$, since must consist of at least one label

Judgmental rules

Judgmental rules

Cut - spawning new process:

Judgmental rules

Cut - spawning new process:

$$\frac{\vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ Cut$$

Judgmental rules

Cut - spawning new process:

$$\frac{x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ Cut$$

Judgmental rules

Cut - spawning new process:

$$\frac{\vdash\; P :: (x : A) \qquad\qquad x : A \vdash\; Q :: (z : C)}{\Delta_1, \Delta_2 \;\vdash\; x \leftarrow P; Q :: (z : C)}\; Cut$$

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ Cut$$

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \; Cut$$

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ Cut$$

split context

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \; Cut$$

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ Cut$$

Identity - forwarding:

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ Cut$$

Identity - forwarding:

$$\frac{}{\vdash \textsf{fwd}\ x\ y :: (x : A)}\ Id$$

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \; Cut$$

Identity - forwarding:

$$\frac{}{\vdash \mathsf{fwd}\; x\; y :: (x : A)} \; Id$$

➡ process offering along x terminates, client henceforth interacts with process offering along y

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \; Cut$$

Identity - forwarding:

$$\frac{}{y : A \vdash \mathsf{fwd}\; x\; y :: (x : A)} \; Id$$

process offering along x terminates, client henceforth interacts with process offering along y

Judgmental rules

Cut - spawning new process:

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)} \; Cut$$

Identity - forwarding:

no orphan providers

$$\frac{}{y : A \vdash \mathsf{fwd}\ x\ y :: (x : A)} \; Id$$

process offering along x terminates, client henceforth interacts with process offering along y

Let’s implement the queue!

We use the formal language SILL used in research papers

The connection to linear logic

The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

The connection to linear logic

➡ if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

$$\frac{\Delta \vdash P :: (x : A)}{\Delta \vdash x.\mathsf{inl}; P :: (x : A \oplus B)} \oplus_{R_1}$$

$$\frac{\Delta \vdash P :: (x : B)}{\Delta \vdash x.\mathsf{inr}; P :: (x : A \oplus B)} \oplus_{R_2}$$

$$\frac{\Delta, x : A \vdash Q_1 :: (z : C) \qquad \Delta, x : B \vdash Q_2 :: (z : C)}{\Delta, x : A \oplus B \vdash \mathsf{case}\ x\ \mathsf{of}(Q_1, Q_2) :: (z : C)} \oplus_L$$

The connection to linear logic

- if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

$$\frac{\Delta \vdash P :: (x : A)}{\Delta \vdash x.\mathsf{inl}; P :: (x : A \oplus B)} \oplus_{R_1} \longrightarrow \frac{\Delta \vdash A}{\Delta \vdash A \oplus B} \oplus_{R_1}$$

$$\frac{\Delta \vdash P :: (x : B)}{\Delta \vdash x.\mathsf{inr}; P :: (x : A \oplus B)} \oplus_{R_2} \longrightarrow \frac{\Delta \vdash B}{\Delta \vdash A \oplus B} \oplus_{R_2}$$

$$\frac{\Delta, x : A \vdash Q_1 :: (z : C) \qquad \Delta, x : B \vdash Q_2 :: (z : C)}{\Delta, x : A \oplus B \vdash \mathsf{case}\, x\, \mathsf{of}(Q_1, Q_2) :: (z : C)} \oplus_L \longrightarrow \frac{\Delta, A \vdash C \qquad \Delta, B \vdash C}{\Delta, A \oplus B \vdash C} \oplus_L$$

The connection to linear logic

if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic

The connection to linear logic

- if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic
- rewrite higher-order channel output with spawn/forward:

The connection to linear logic

- if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic
- rewrite higher-order channel output with spawn/forward:

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \mathsf{send}\ x\ y; P :: (x : A \otimes B)} \otimes_R$$

The connection to linear logic

- if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic
- rewrite higher-order channel output with spawn/forward:

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \mathsf{send}\ x\ y; P :: (x : A \otimes B)} \otimes_R$$

$$\Downarrow$$

$$\frac{y : A \vdash \mathsf{fwd}\ z\ y :: (z : A) \qquad \Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \mathsf{send}\ x\ (z \leftarrow \mathsf{fwd}\ z\ y); P :: (x : A \otimes B)} \otimes_R$$

The connection to linear logic

- if we erase process terms in typing rules, we get left and right rules of intuitionistic linear logic
- rewrite higher-order channel output with spawn/forward:

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \mathsf{send}\ x\ y; P :: (x : A \otimes B)} \otimes_R$$

$$\Downarrow$$

$$\frac{y : A \vdash \mathsf{fwd}\ z\ y :: (z : A) \qquad \Delta \vdash P :: (x : B)}{\Delta, y : A \vdash \mathsf{send}\ x\ (z \leftarrow \mathsf{fwd}\ z\ y); P :: (x : A \otimes B)} \otimes_R \quad \longrightarrow \quad \frac{A \vdash A \qquad \Delta \vdash B}{\Delta, A \vdash A \otimes B} \otimes_R$$

Curry-Howard correspondence

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic: Type theory:

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	session types

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	session types
proofs	

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	session types
proofs	programs

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	session types
proofs	programs
cut reduction	

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	session types
proofs	programs
cut reduction	communication

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	session types
proofs	programs
cut reduction	communication

Luis Caires and Frank Pfenning. Session types as intuitionistic linear propositions. CONCUR, 2010.

Curry-Howard correspondence

Correspondence between linear logic and session-typed pi-calculus

Logic:	Type theory:
linear propositions	session types
proofs	programs
cut reduction	communication

Luis Caires and Frank Pfenning. Session types as intuitionistic linear propositions. CONCUR, 2010.

Philip Wadler. Propositions as sessions. ICFP, 2012.

Benefits of linear logic for programming

Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathit{weaken} \qquad\qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathit{contract}$$

Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathit{weaken} \qquad\qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathit{contract}$$

“drop resource”

Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathit{weaken} \qquad\qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathit{contract}$$

"drop resource"   "duplicate resource"

Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathit{weaken} \qquad\qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathit{contract}$$

"drop resource" "duplicate resource"

→ without weakening, every provider has at least one client

Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathit{weaken} \qquad\qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathit{contract}$$

"drop resource" "duplicate resource"

→ without weakening, every provider has at least one client

→ without contraction, every provider has at most one client

Benefits of linear logic for programming

Linear logic is a substructural logic because it rejects the structural rules of weakening and contraction:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \textit{weaken} \qquad\qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \textit{contract}$$

"drop resource" "duplicate resource"

- without weakening, every provider has at least one client
- without contraction, every provider has at most one client
- thus, every provider has exactly one client

Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

Benefits of linear logic for programming

Let’s identify absence of weakening and contraction in our rules:

no resources can be dropped

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

no resources
can be dropped

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

every provider has
at least one client

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ \mathit{Cut}$$

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ \mathit{Cut}$$

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ \mathit{Cut}$$

no resources
duplicated

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \textsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

no aliases
created

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ \mathit{Cut}$$

no resources
duplicated

Benefits of linear logic for programming

Let's identify absence of weakening and contraction in our rules:

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}_R$$

no aliases created

$$\frac{\Delta_1 \vdash P :: (x : A) \qquad \Delta_2, x : A \vdash Q :: (z : C)}{\Delta_1, \Delta_2 \vdash x \leftarrow P; Q :: (z : C)}\ \mathit{Cut}$$

no resources duplicated

every provider has at most one client

Benefits of linear logic for programming

Benefits of linear logic for programming

linear logic session types turn run-time process graph into a tree

Benefits of linear logic for programming

- linear logic session types turn run-time process graph into a tree
- for intuitionistic linear logic session types, tree is directed

Benefits of linear logic for programming

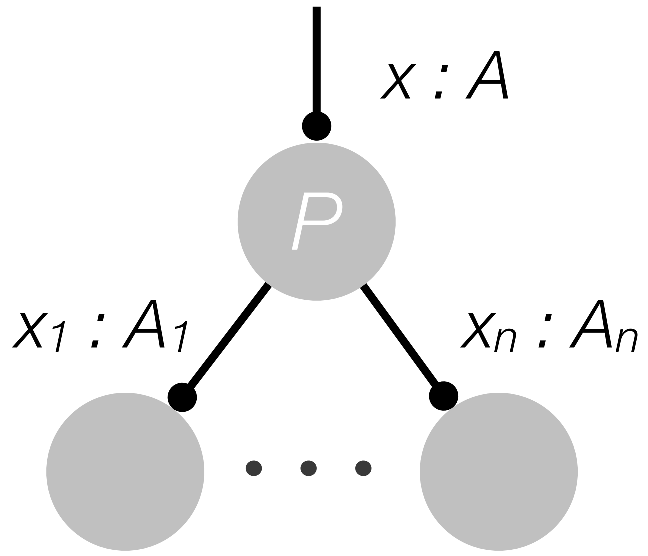
- linear logic session types turn run-time process graph into a tree
- for intuitionistic linear logic session types, tree is directed

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)$$

Benefits of linear logic for programming

- linear logic session types turn run-time process graph into a tree
- for intuitionistic linear logic session types, tree is directed

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)$$

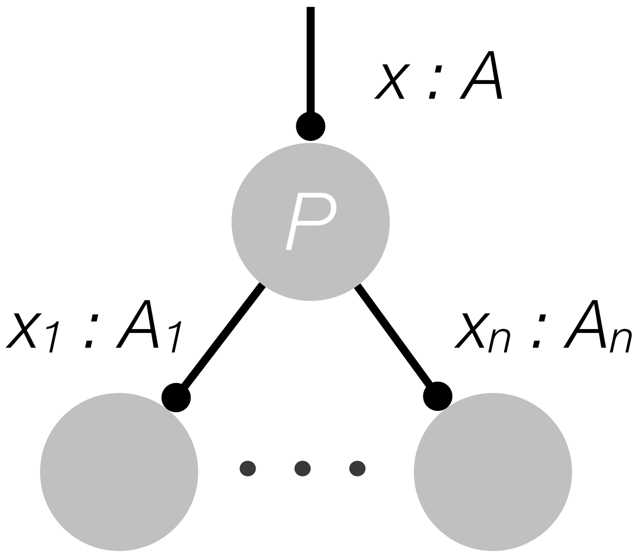


Benefits of linear logic for programming

- linear logic session types turn run-time process graph into a tree
- for intuitionistic linear logic session types, tree is directed

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)$$

parent: client



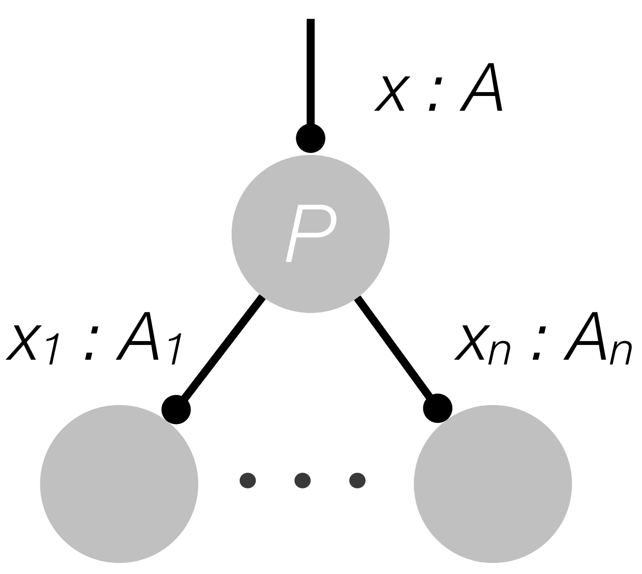
Benefits of linear logic for programming

- linear logic session types turn run-time process graph into a tree
- for intuitionistic linear logic session types, tree is directed

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)$$

parent: client

child: provider



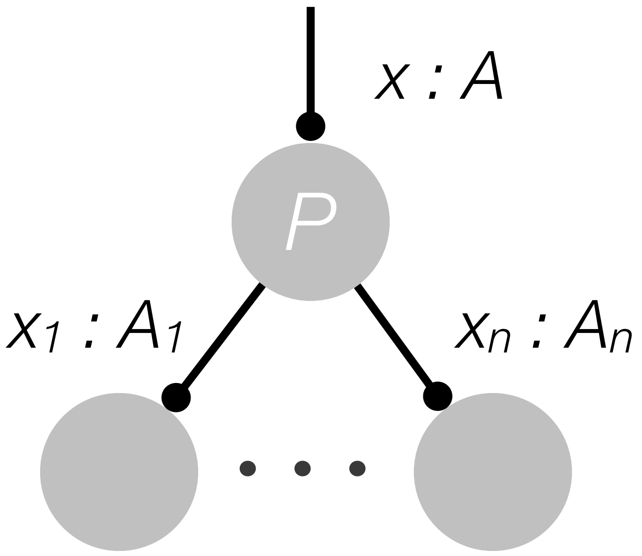
Benefits of linear logic for programming

- linear logic session types turn run-time process graph into a tree
- for intuitionistic linear logic session types, tree is directed

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)$$

parent: client

child: provider



- we will use directedness for deadlock-freedom

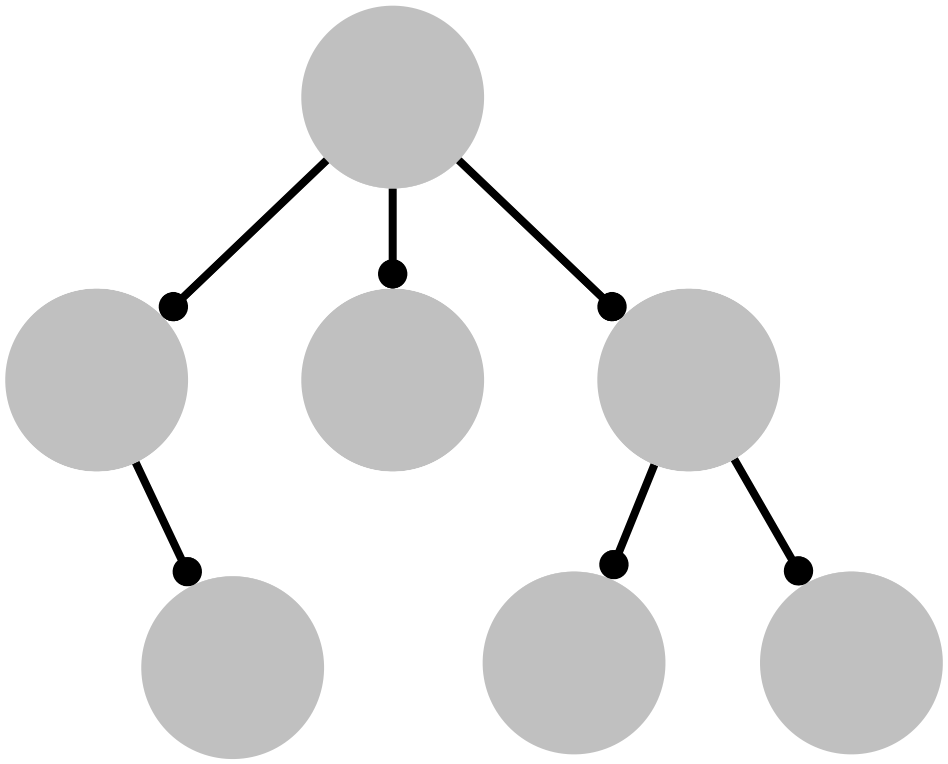
Benefits of linear logic for programming

Benefits of linear logic for programming

- type safety holds easily:

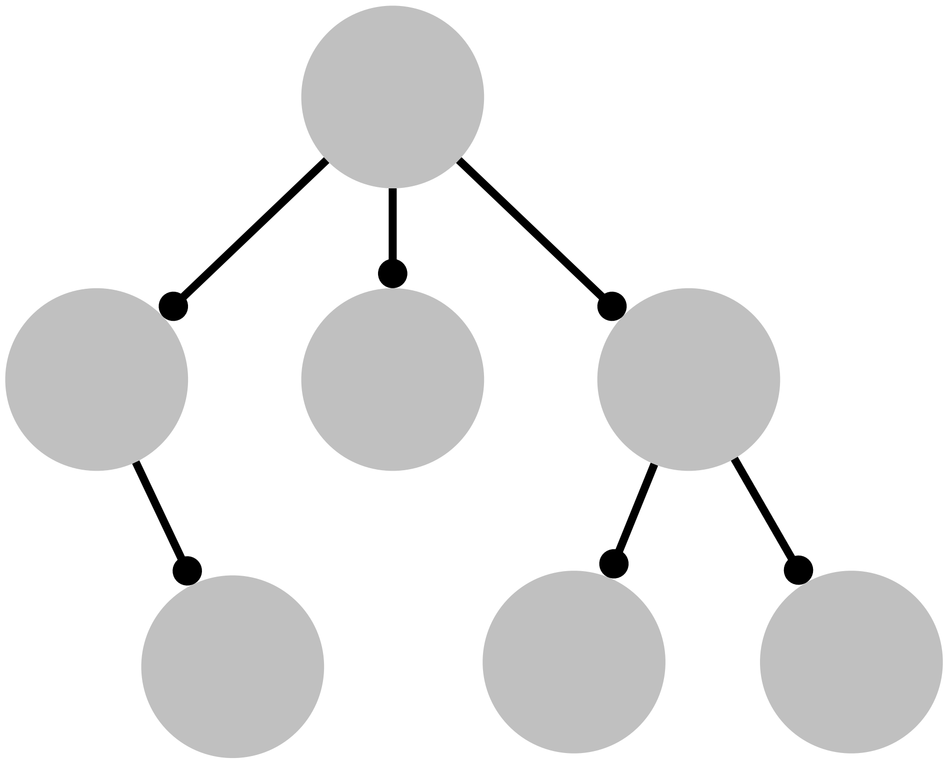
Benefits of linear logic for programming

type safety holds easily:



Benefits of linear logic for programming

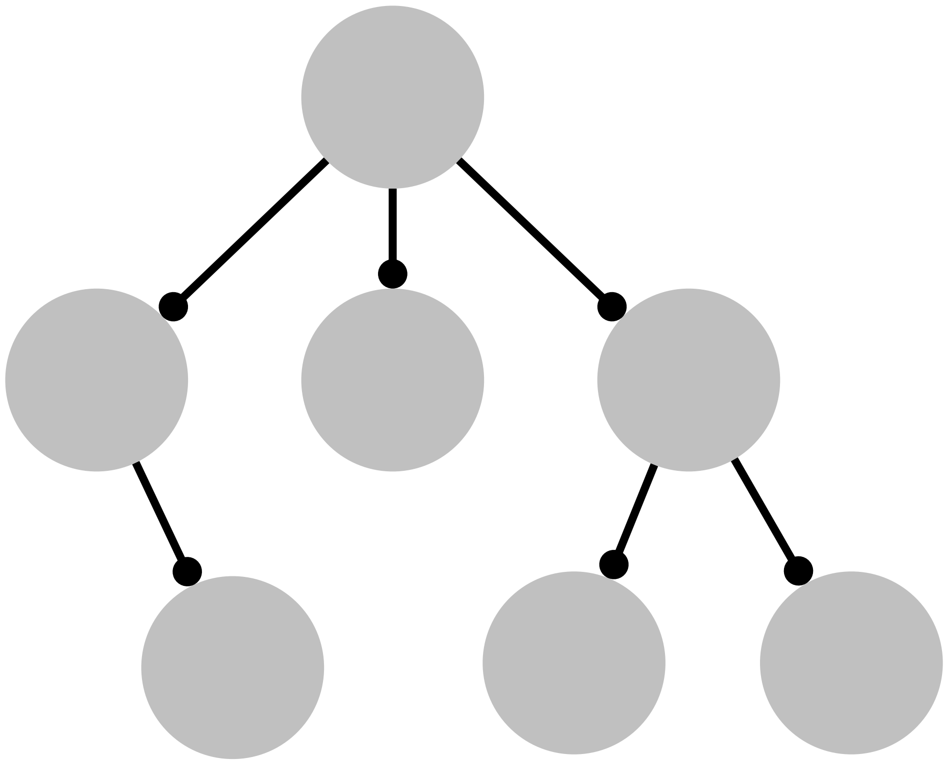
type safety holds easily:



Preservation (a.k.a., session fidelity)

Benefits of linear logic for programming

type safety holds easily:

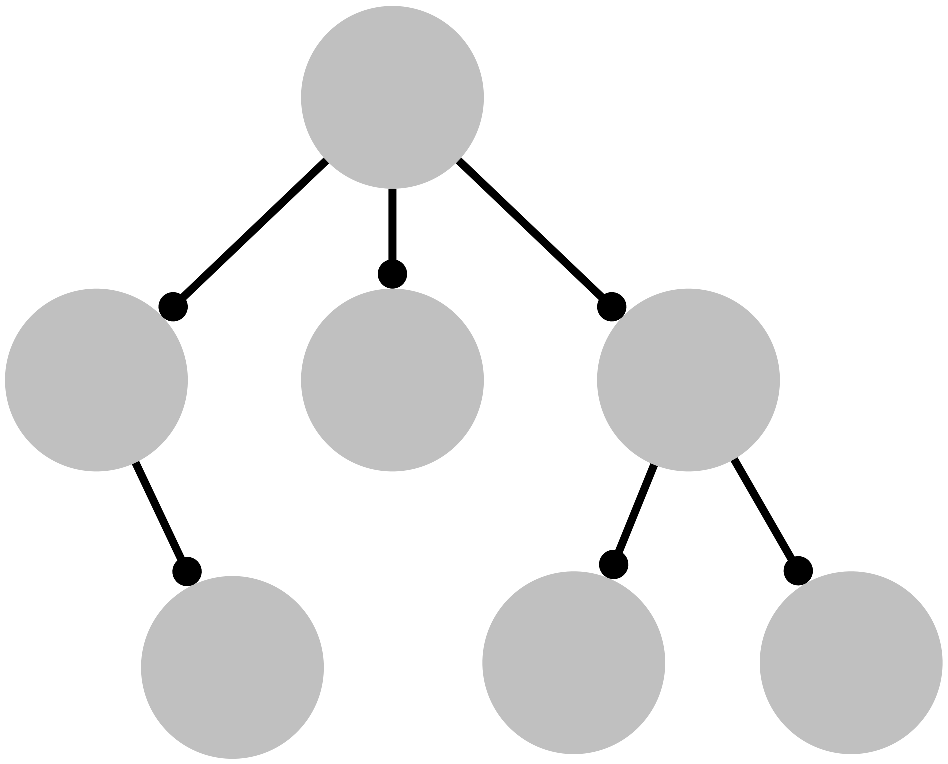


Preservation (a.k.a., session fidelity)

- every provider has a unique client

Benefits of linear logic for programming

type safety holds easily:

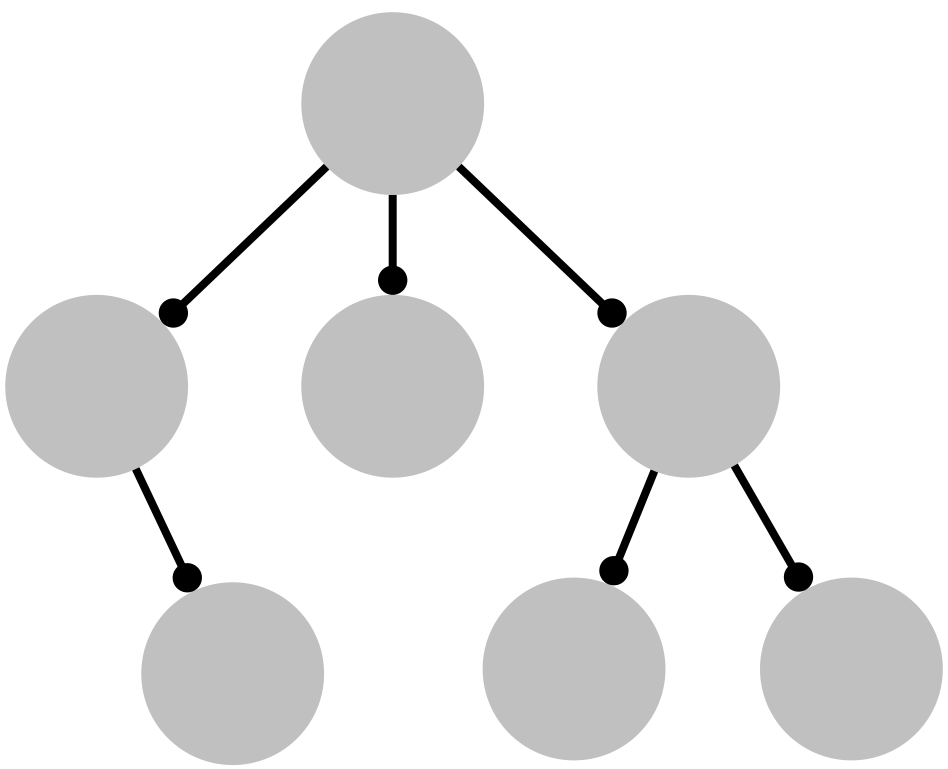


Preservation (a.k.a., session fidelity) ✓

- every provider has a unique client

Benefits of linear logic for programming

type safety holds easily:



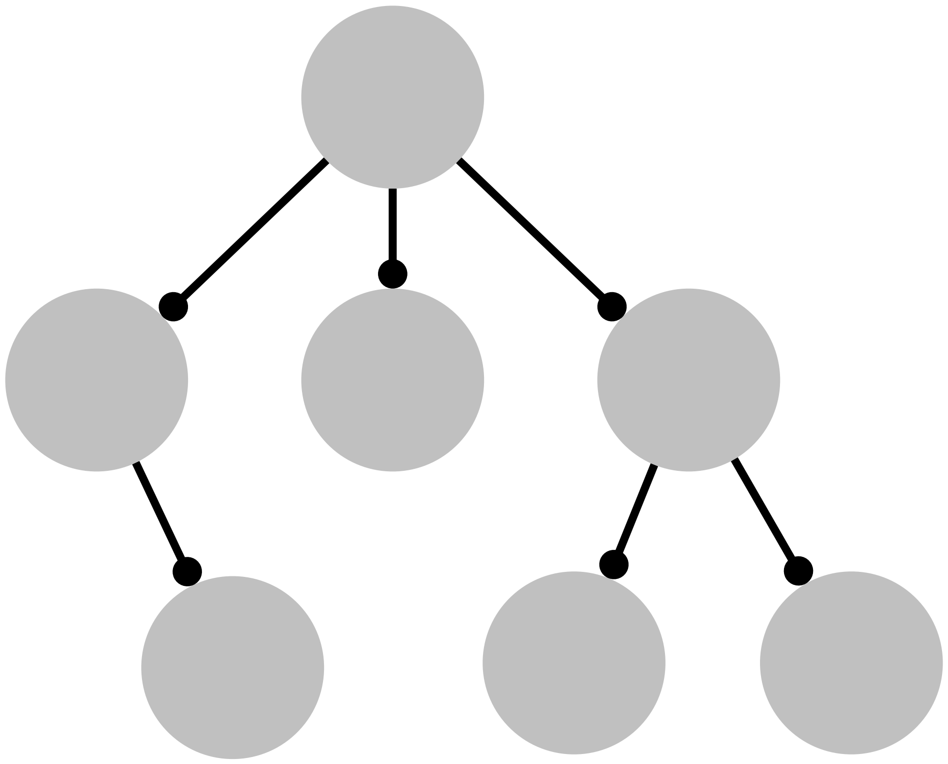
Preservation (a.k.a., session fidelity) ✓

- every provider has a unique client

Progress (a.k.a., deadlock-freedom)

Benefits of linear logic for programming

type safety holds easily:



Preservation (a.k.a., session fidelity) ✓

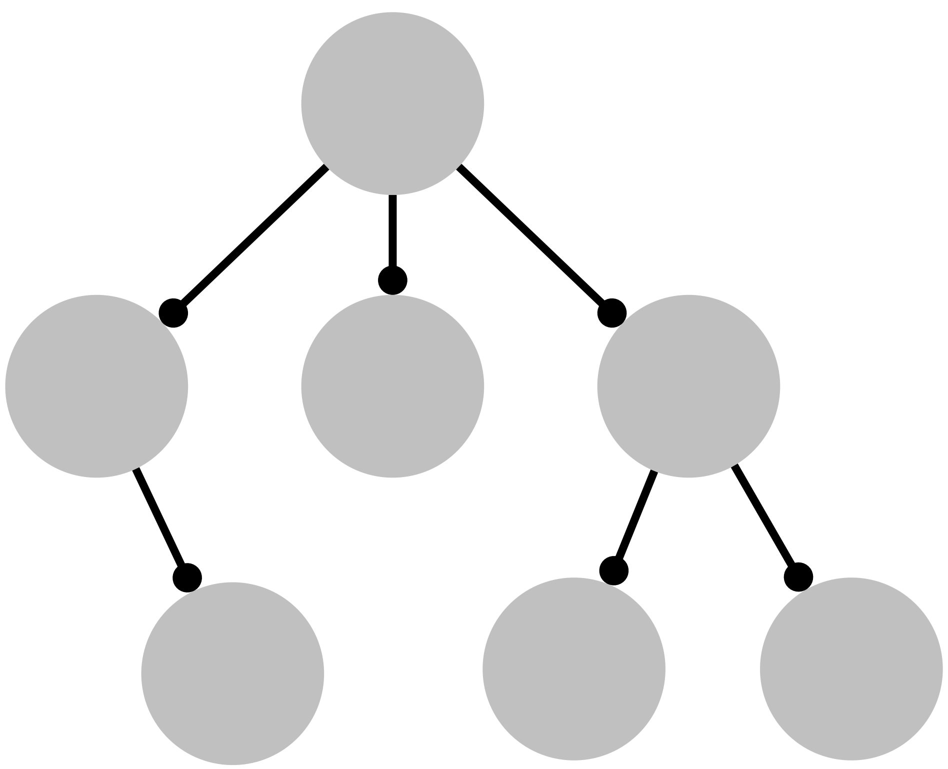
- every provider has a unique client

Progress (a.k.a., deadlock-freedom)

- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not

Benefits of linear logic for programming

➡ **type safety holds easily:**

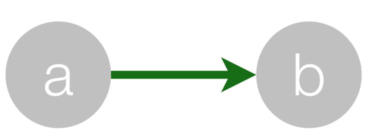


Preservation (a.k.a., session fidelity) ✓

- every provider has a unique client

Progress (a.k.a., deadlock-freedom)

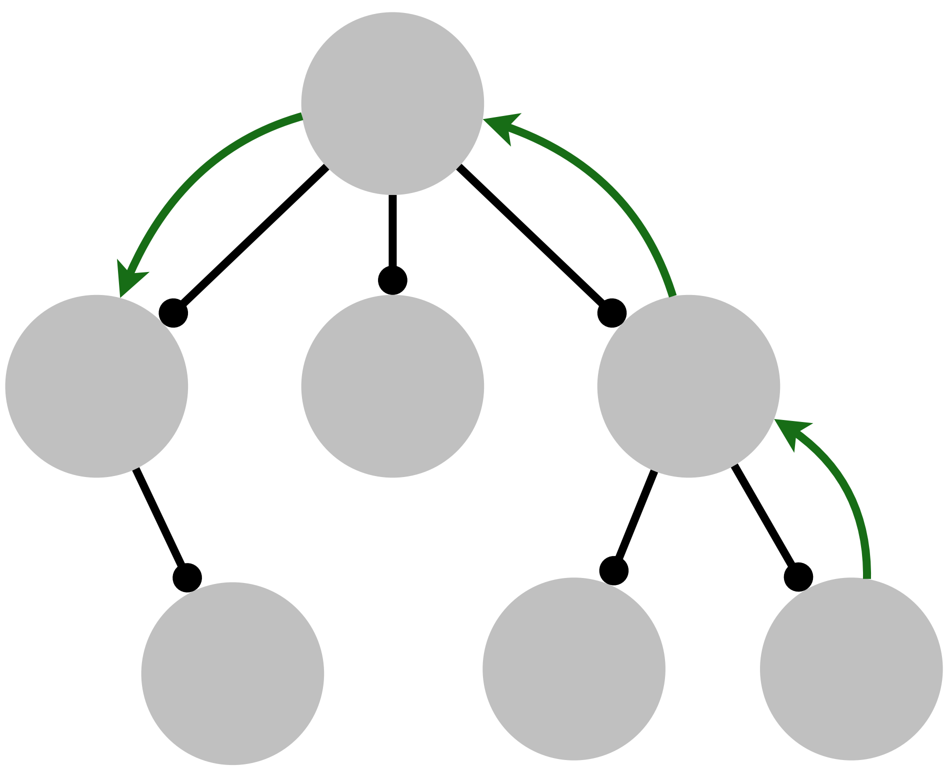
- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not



“a waits for b”

Benefits of linear logic for programming

➡ type safety holds easily:

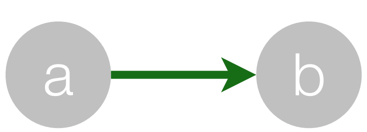


Preservation (a.k.a., session fidelity) ✓

- every provider has a unique client

Progress (a.k.a., deadlock-freedom)

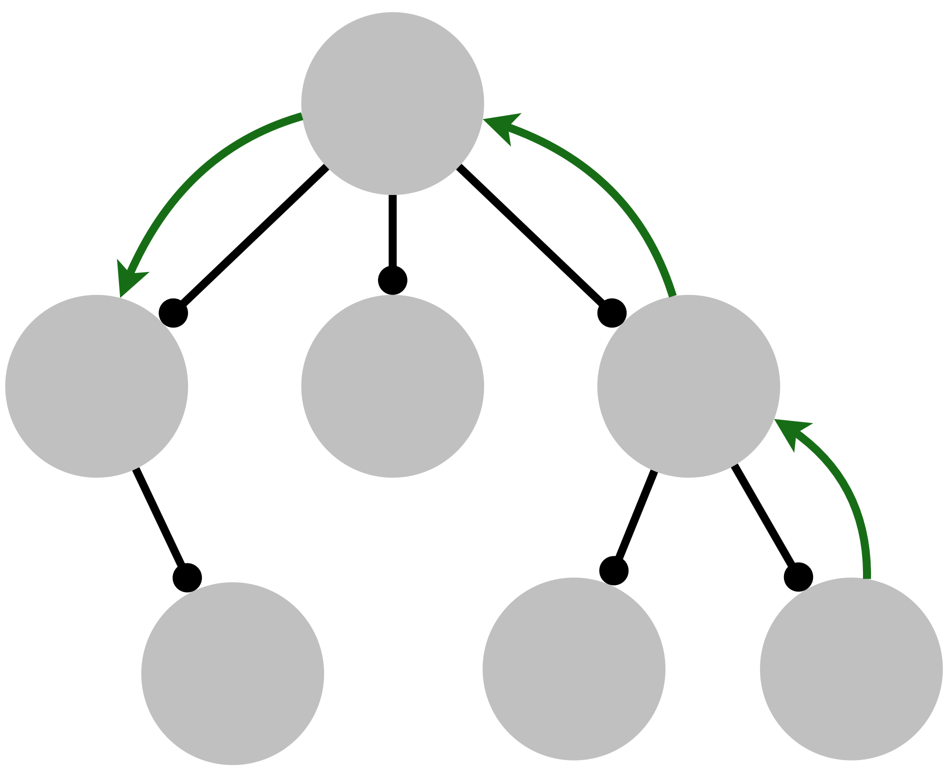
- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not

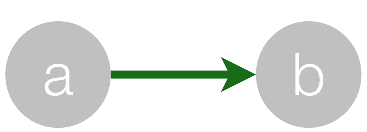


“a waits for b”

Benefits of linear logic for programming

type safety holds easily:





“a waits for b”

Preservation (a.k.a., session fidelity) ✓

- every provider has a unique client

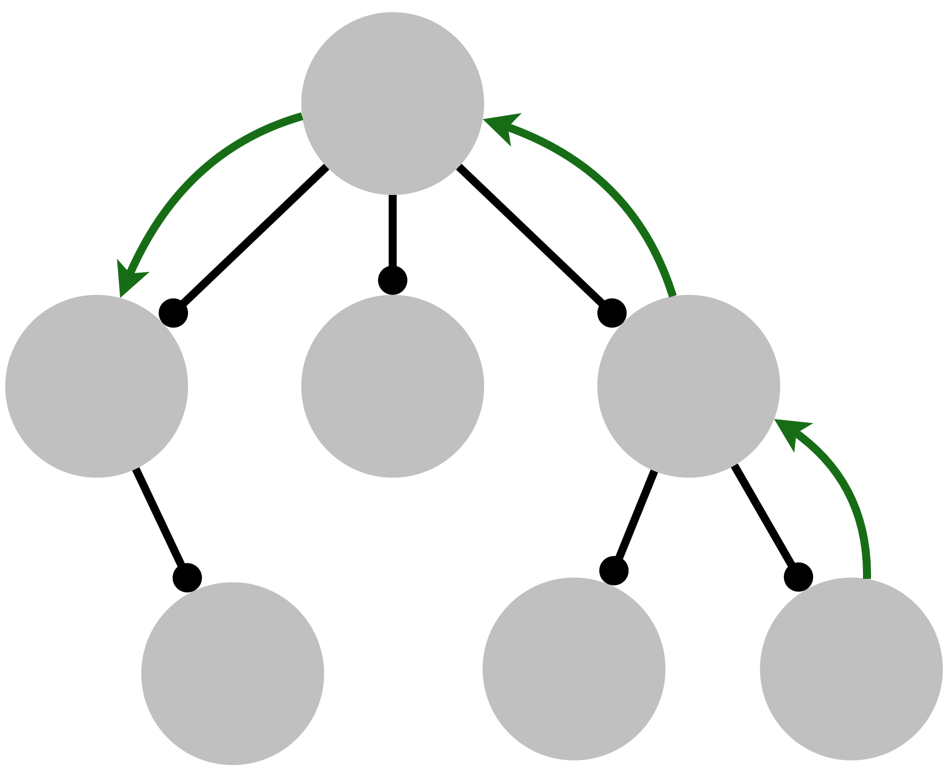
Progress (a.k.a., deadlock-freedom)

- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not

green arrows can only go along edges, thus cannot form a cycle

Benefits of linear logic for programming

type safety holds easily:

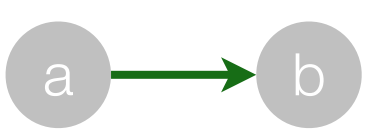


Preservation (a.k.a., session fidelity) ✓

- every provider has a unique client

Progress (a.k.a., deadlock-freedom) ✓

- 2 possible threats to progress:
  - provider ready to synchronize, client not
  - client ready to synchronize, provider not



“a waits for b”

green arrows can only go along edges, thus cannot form a cycle

Type safety formalized

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

→ let's define the dynamics

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ let's define the dynamics

Multiset rewriting rules:

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ **let's define the dynamics**

Multiset rewriting rules:

$$\begin{array}{l}\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q') \\ \longrightarrow\ \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')\end{array}$$

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ let's define the dynamics

Multiset rewriting rules:

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q')$$
$$\longrightarrow\ \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

➡ rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

→ let's define the dynamics

Multiset rewriting rules:

before rewrite

$$\text{proc}(a, P\langle a\rangle; P'), \text{proc}(c, Q\langle a\rangle; Q')$$
$$\longrightarrow\ \text{proc}(a, P'), \text{proc}(c, Q')$$

→ rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ let's define the dynamics

Multiset rewriting rules:

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q')$$
$$\longrightarrow\ \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

➡ rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ **let's define the dynamics**

Multiset rewriting rules:

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q')$$
$$\longrightarrow \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

after rewrite

➡ **rewrite process tree only describing what changes**

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ **let's define the dynamics**

Multiset rewriting rules:

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q') \\ \longrightarrow \ \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

➡ **rewrite process tree only describing what changes**

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let's define the dynamics

Multiset rewriting rules:

provider

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q')$$
$$\longrightarrow \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ let's define the dynamics

Multiset rewriting rules:

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q') \\ \longrightarrow \ \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

➡ rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let's define the dynamics

Multiset rewriting rules:

client

$$\text{proc}(a, P\langle a\rangle; P'), \text{proc}(c, Q\langle a\rangle; Q')$$
$$\longrightarrow \text{proc}(a, P'), \text{proc}(c, Q')$$

rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let's define the dynamics

Multiset rewriting rules:

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q')$$
$$\longrightarrow\ \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

let's define the dynamics

Multiset rewriting rules:

offering channel

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q') \longrightarrow \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

rewrite process tree only describing what changes

Type safety formalized

Type safety expresses coherence between statics (type system) and dynamics of a language

➡ **let's define the dynamics**

Multiset rewriting rules:

code being executed

offering channel

$$\mathsf{proc}(a, P\langle a\rangle; P'), \mathsf{proc}(c, Q\langle a\rangle; Q') \longrightarrow \mathsf{proc}(a, P'), \mathsf{proc}(c, Q')$$

➡ **rewrite process tree only describing what changes**

Dynamics

Selected rules:

Dynamics

Selected rules:

$$(\text{D-}\otimes) \quad \mathsf{proc}(a, \mathsf{send}\ a\ b; P), \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a; Q_y)$$
$$\longrightarrow\ \mathsf{proc}(a, P), \mathsf{proc}(c, [b/y]\, Q_y)$$

Dynamics

Selected rules:

$$(\text{D-}\otimes) \quad \mathsf{proc}(a, \mathsf{send}\ a\ b; P), \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a; Q_y)$$
$$\longrightarrow\ \mathsf{proc}(a, P), \mathsf{proc}(c, [b/y]\, Q_y)$$

$$(\text{D-}\&) \quad \mathsf{proc}(a, \mathsf{case}\, a\, \mathsf{of}\, \overline{l \Rightarrow P}), \mathsf{proc}(c, a.l_k; Q)$$
$$\longrightarrow\ \mathsf{proc}(a, P_k), \mathsf{proc}(c, Q)$$

Dynamics

Selected rules:

$$(\text{D-}\otimes) \quad \mathsf{proc}(a, \mathsf{send}\ a\ b; P), \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a; Q_y)$$
$$\longrightarrow \ \mathsf{proc}(a, P), \mathsf{proc}(c, [b/y]\, Q_y)$$

$$(\text{D-}\&) \quad \mathsf{proc}(a, \mathsf{case}\, a\, \mathsf{of}\, \overline{l \Rightarrow P}), \mathsf{proc}(c, a.l_k; Q)$$
$$\longrightarrow \ \mathsf{proc}(a, P_k), \mathsf{proc}(c, Q)$$

$$(\text{D-}\mathbf{1}) \quad \mathsf{proc}(a, \mathsf{close}\ a), \mathsf{proc}(c, \mathsf{wait}\ a; Q)$$
$$\longrightarrow \ \mathsf{proc}(c, Q)$$

Dynamics

Selected rules:

$$(\text{D-}\otimes) \quad \mathsf{proc}(a, \mathsf{send}\ a\ b; P), \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a; Q_y)$$
$$\longrightarrow\ \mathsf{proc}(a, P), \mathsf{proc}(c, [b/y]\, Q_y)$$

$$(\text{D-}\&) \quad \mathsf{proc}(a, \mathsf{case}\, a\, \mathsf{of}\, \overline{l \Rightarrow P}), \mathsf{proc}(c, a.l_k; Q)$$
$$\longrightarrow\ \mathsf{proc}(a, P_k), \mathsf{proc}(c, Q)$$

$$(\text{D-}\mathbf{1}) \quad \mathsf{proc}(a, \mathsf{close}\ a), \mathsf{proc}(c, \mathsf{wait}\ a; Q)$$
$$\longrightarrow\ \mathsf{proc}(c, Q)$$

$$(\text{D-}Cut) \quad \mathsf{proc}(c, x \leftarrow P_x; Q_x)$$
$$\longrightarrow\ \mathsf{proc}(a, [a/x]\, P_x), \mathsf{proc}(c, [a/x]\, Q_x) \qquad (\text{a fresh})$$

Dynamics

Selected rules:

$$(\text{D-}\otimes) \quad \mathsf{proc}(a, \mathsf{send}\ a\ b; P), \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a; Q_y)$$
$$\longrightarrow\ \mathsf{proc}(a, P), \mathsf{proc}(c, [b/y]\, Q_y)$$

$$(\text{D-}\&) \quad \mathsf{proc}(a, \mathsf{case}\, a\, \mathsf{of}\, \overline{l \Rightarrow P}), \mathsf{proc}(c, a.l_k; Q)$$
$$\longrightarrow\ \mathsf{proc}(a, P_k), \mathsf{proc}(c, Q)$$

$$(\text{D-}\mathbf{1}) \quad \mathsf{proc}(a, \mathsf{close}\ a), \mathsf{proc}(c, \mathsf{wait}\ a; Q)$$
$$\longrightarrow\ \mathsf{proc}(c, Q)$$

$$(\text{D-}\mathit{Cut}) \quad \mathsf{proc}(c, x \leftarrow P_x; Q_x)$$
$$\longrightarrow\ \mathsf{proc}(a, [a/x]\, P_x), \mathsf{proc}(c, [a/x]\, Q_x) \qquad (\text{a fresh})$$

$$(\text{D-}\mathit{Id}) \quad \mathsf{proc}(a, \mathsf{fwd}\ a\ b)$$
$$\longrightarrow\ (\text{a} = \text{b})$$

Dynamics

Selected rules:

$$(\text{D-}\otimes) \quad \mathsf{proc}(a, \mathsf{send}\ a\ b; P), \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a; Q_y)$$
$$\longrightarrow \mathsf{proc}(a, P), \mathsf{proc}(c, [b/y]\, Q_y)$$

synchronous dynamic

$$(\text{D-}\&) \quad \mathsf{proc}(a, \mathsf{case}\, a\, \mathsf{of}\, \overline{l \Rightarrow P}), \mathsf{proc}(c, a.l_k; Q)$$
$$\longrightarrow \mathsf{proc}(a, P_k), \mathsf{proc}(c, Q)$$

$$(\text{D-}\mathbf{1}) \quad \mathsf{proc}(a, \mathsf{close}\ a), \mathsf{proc}(c, \mathsf{wait}\ a; Q)$$
$$\longrightarrow \mathsf{proc}(c, Q)$$

$$(\text{D-}\mathit{Cut}) \quad \mathsf{proc}(c, x \leftarrow P_x; Q_x)$$
$$\longrightarrow \mathsf{proc}(a, [a/x]\, P_x), \mathsf{proc}(c, [a/x]\, Q_x) \qquad (\text{a fresh})$$

$$(\text{D-}\mathit{Id}) \quad \mathsf{proc}(a, \mathsf{fwd}\ a\ b)$$
$$\longrightarrow (\text{a} = \text{b})$$

Dynamics

Selected rules:

$$(\text{D-}\otimes)\quad \mathsf{proc}(a, \mathsf{send}\ a\ b; P), \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a; Q_y) \longrightarrow \mathsf{proc}(a, P), \mathsf{proc}(c, [b/y]\, Q_y)$$

$$(\text{D-}\&)\quad \mathsf{proc}(a, \mathsf{case}\ a\ \mathsf{of}\ \overline{l \Rightarrow P}), \mathsf{proc}(c, a.l_k; Q) \longrightarrow \mathsf{proc}(a, P_k), \mathsf{proc}(c, Q)$$

$$(\text{D-}\mathbf{1})\quad \mathsf{proc}(a, \mathsf{close}\ a), \mathsf{proc}(c, \mathsf{wait}\ a; Q) \longrightarrow \mathsf{proc}(c, Q)$$

$$(\text{D-}Cut)\quad \mathsf{proc}(c, x \leftarrow P_x; Q_x) \longrightarrow \mathsf{proc}(a, [a/x]\, P_x), \mathsf{proc}(c, [a/x]\, Q_x) \qquad (\text{a fresh})$$

$$(\text{D-}Id)\quad \mathsf{proc}(a, \mathsf{fwd}\ a\ b) \longrightarrow (\text{a} = \text{b})$$

synchronous dynamic

both send and receive are blocking

Dynamics

Selected rules:

(D-$\otimes$) $\text{proc}(a, \text{send } a\ b; P), \text{proc}(c, y \leftarrow \text{recv } a; Q_y)$
$\longrightarrow\ \text{proc}(a, P), \text{proc}(c, [b/y]\, Q_y)$

synchronous dynamic

(D-&) $\text{proc}(a, \text{case } a \text{ of } \overline{l \Rightarrow P}), \text{proc}(c, a.l_k; Q)$
$\longrightarrow\ \text{proc}(a, P_k), \text{proc}(c, Q)$

both send and receive are blocking

(D-$\mathbf{1}$) $\text{proc}(a, \text{close } a), \text{proc}(c, \text{wait } a; Q)$
$\longrightarrow\ \text{proc}(c, Q)$

(D-Cut) $\text{proc}(c, x \leftarrow P_x; Q_x)$
$\longrightarrow\ \text{proc}(a, [a/x]\, P_x), \text{proc}(c, [a/x]\, Q_x)$ (a fresh)

(D-Id) $\text{proc}(a, \text{fwd } a\ b)$
$\longrightarrow\ (\text{a} = \text{b})$

asynchronous semantics: spawns off messages and links them with forward

Configuration typing

Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

$$\frac{}{\vDash (\cdot) :: (\cdot)}$$

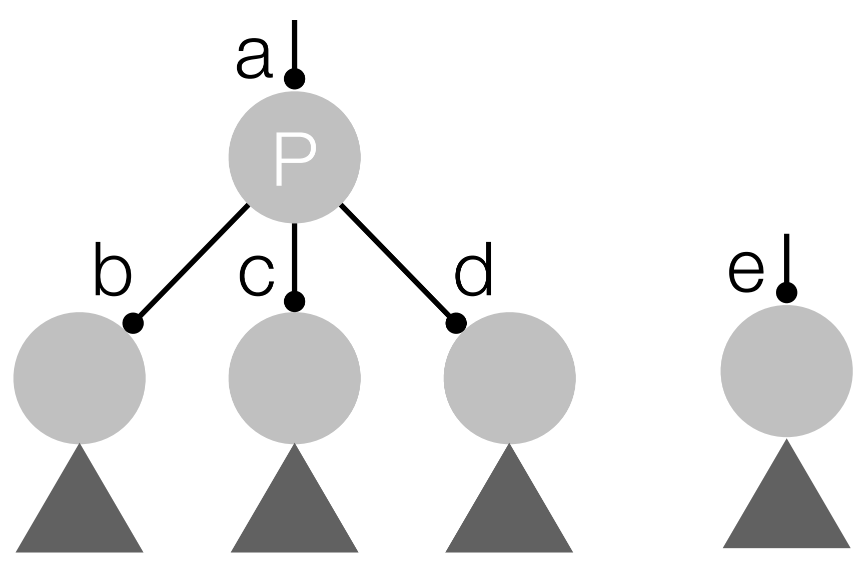
$$\frac{\Gamma \vDash \Omega :: \Delta_1, \Delta_2 \qquad \Delta_1 \vdash P_a :: (a : A)}{\vDash \Omega, \mathsf{proc}(a, P_a) :: (\Delta_2, a : A)}$$

Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

$$\frac{}{\vDash (\cdot) :: (\cdot)}$$

$$\frac{\Gamma \vDash \Omega :: \Delta_1, \Delta_2 \qquad \Delta_1 \vdash P_a :: (a : A)}{\vDash \Omega, \mathsf{proc}(a, P_a) :: (\Delta_2, a : A)}$$

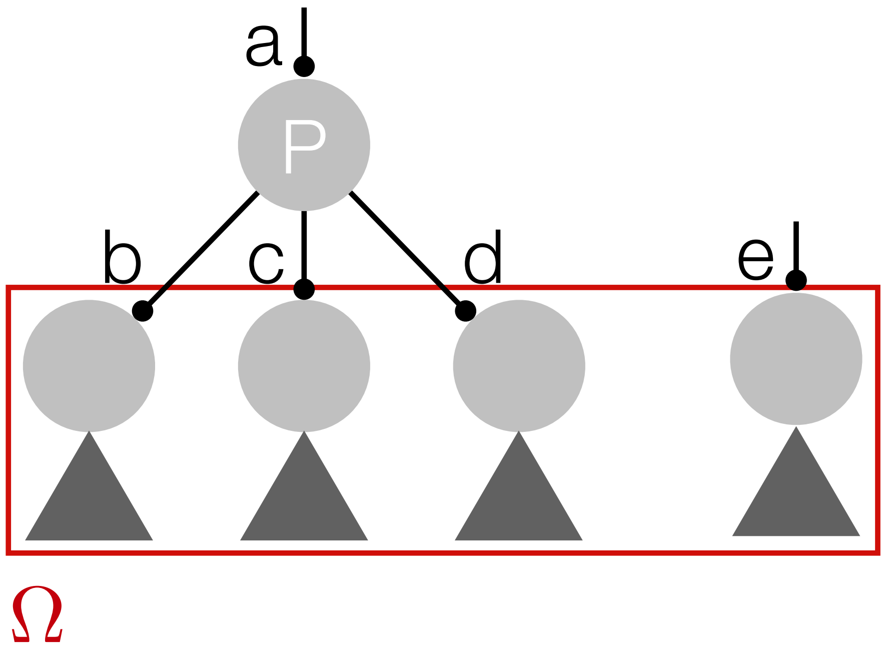


Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

$$\frac{}{\vDash (\cdot) :: (\cdot)}$$

$$\frac{\Gamma \vDash \Omega :: \Delta_1, \Delta_2 \qquad \Delta_1 \vdash P_a :: (a : A)}{\vDash \Omega, \mathsf{proc}(a, P_a) :: (\Delta_2, a : A)}$$

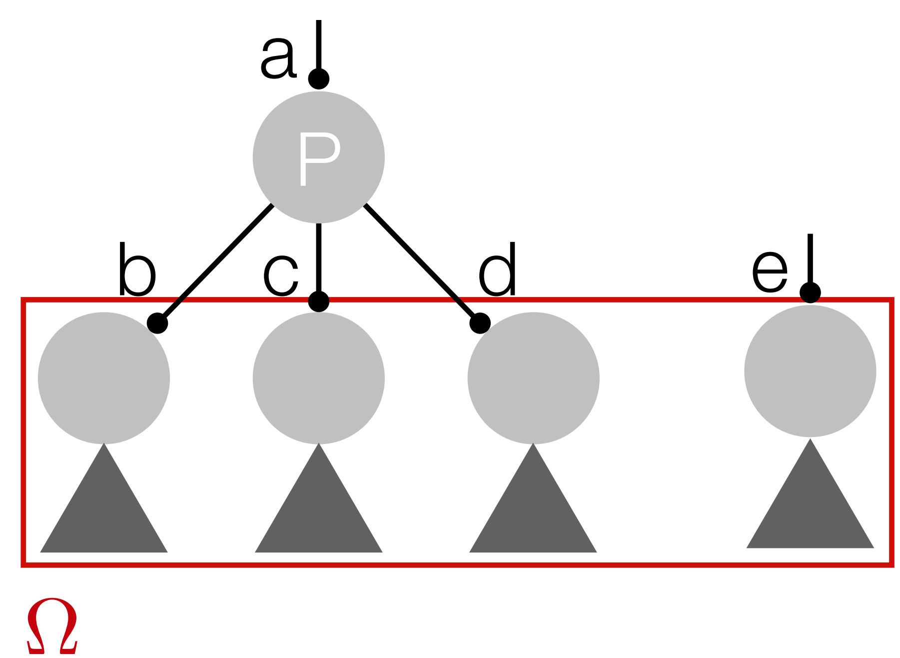


Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

$$\frac{}{\vDash (\cdot) :: (\cdot)}$$

$$\frac{\Gamma \vDash \Omega :: \Delta_1, \Delta_2 \qquad \Delta_1 \vdash P_a :: (a : A)}{\vDash \Omega, \mathsf{proc}(a, P_a) :: (\Delta_2, a : A)}$$



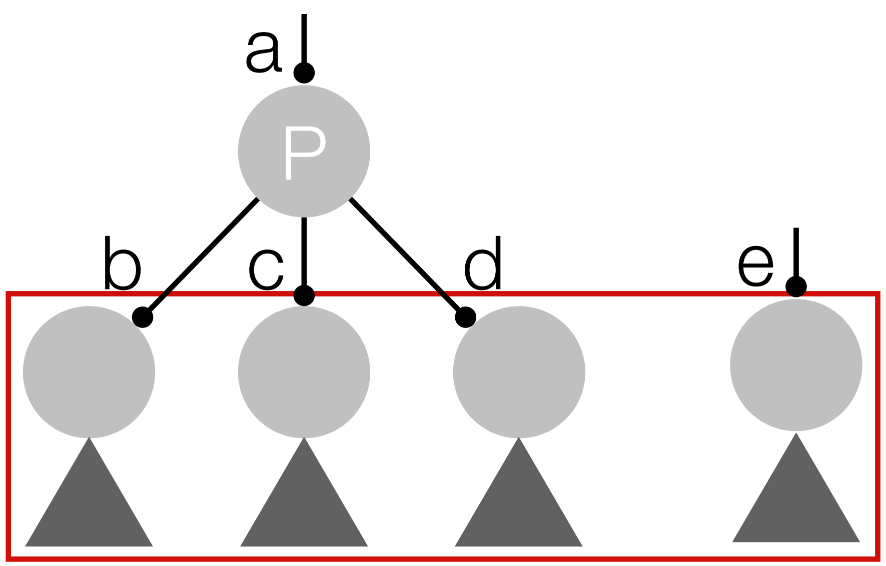
$\Delta_1 = \mathsf{b}, \mathsf{c}, \mathsf{d}$

Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

$$\overline{\vDash (\cdot) :: (\cdot)}$$

$$\frac{\Gamma \vDash \Omega :: \Delta_1, \Delta_2 \qquad \Delta_1 \vdash P_a :: (a : A)}{\vDash \Omega, \mathsf{proc}(a, P_a) :: (\Delta_2, a : A)}$$



Ω

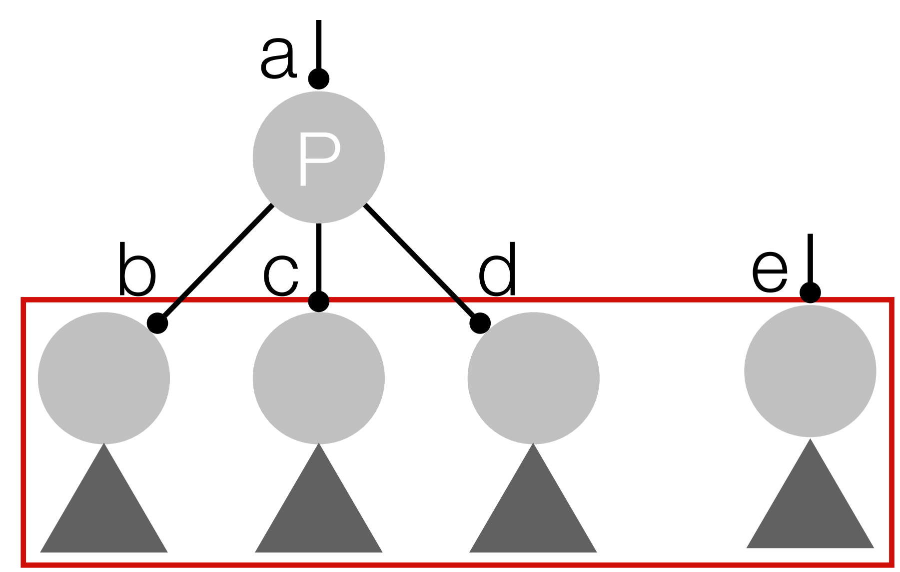
$\Delta_1 = \mathsf{b, c, d} \quad \Delta_2 = \mathsf{e}$

Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

$$\frac{}{\vDash (\cdot) :: (\cdot)}$$

$$\frac{\Gamma \vDash \Omega :: \Delta_1, \Delta_2 \qquad \Delta_1 \vdash P_a :: (a : A)}{\vDash \Omega, \mathsf{proc}(a, P_a) :: (\Delta_2, a : A)}$$



Ω

$\Delta_1 = \mathsf{b, c, d} \quad \Delta_2 = \mathsf{e}$

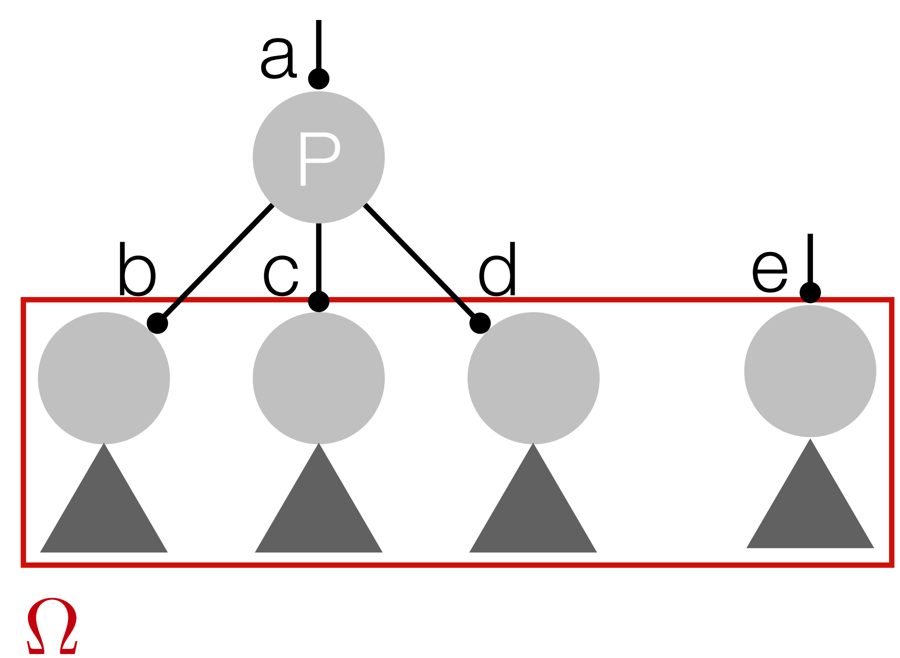
typing imposes forest structure and tree structure at top level

Configuration typing

In addition to typing process terms, we must type the run-time configuration of processes (a.k.a. heap typing)

$$\frac{}{\vDash (\cdot) :: (\cdot)}$$

$$\frac{\Gamma \vDash \Omega :: \Delta_1, \Delta_2 \qquad \Delta_1 \vdash P_a :: (a : A)}{\vDash \Omega, \mathsf{proc}(a, P_a) :: (\Delta_2, a : A)}$$



$\Delta_1 = \mathsf{b}, \mathsf{c}, \mathsf{d} \quad \Delta_2 = \mathsf{e}$

- typing imposes forest structure and tree structure at top level
- a closed program offers a session of type $\mathbf{1}$, the top-level "main" process

Preservation and progress

Preservation and progress

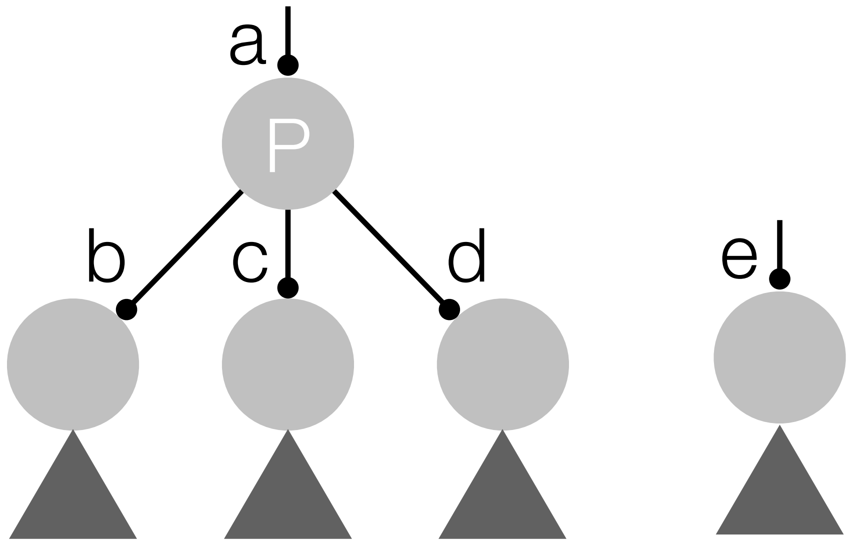
Theorem (Preservation)**.** *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$*, then* $\vDash \Omega' :: \Delta$.

Preservation and progress

Theorem (Preservation)**.** *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.

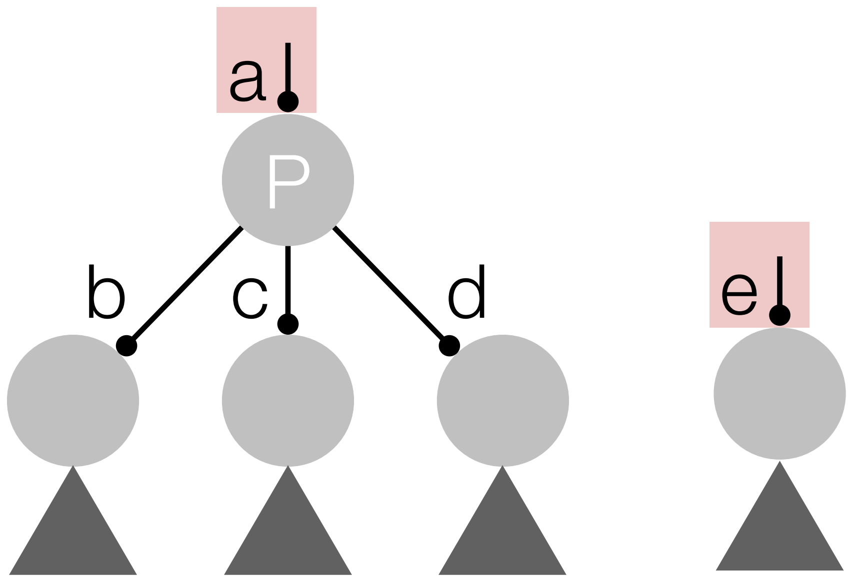
Preservation and progress

Theorem (Preservation). *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.



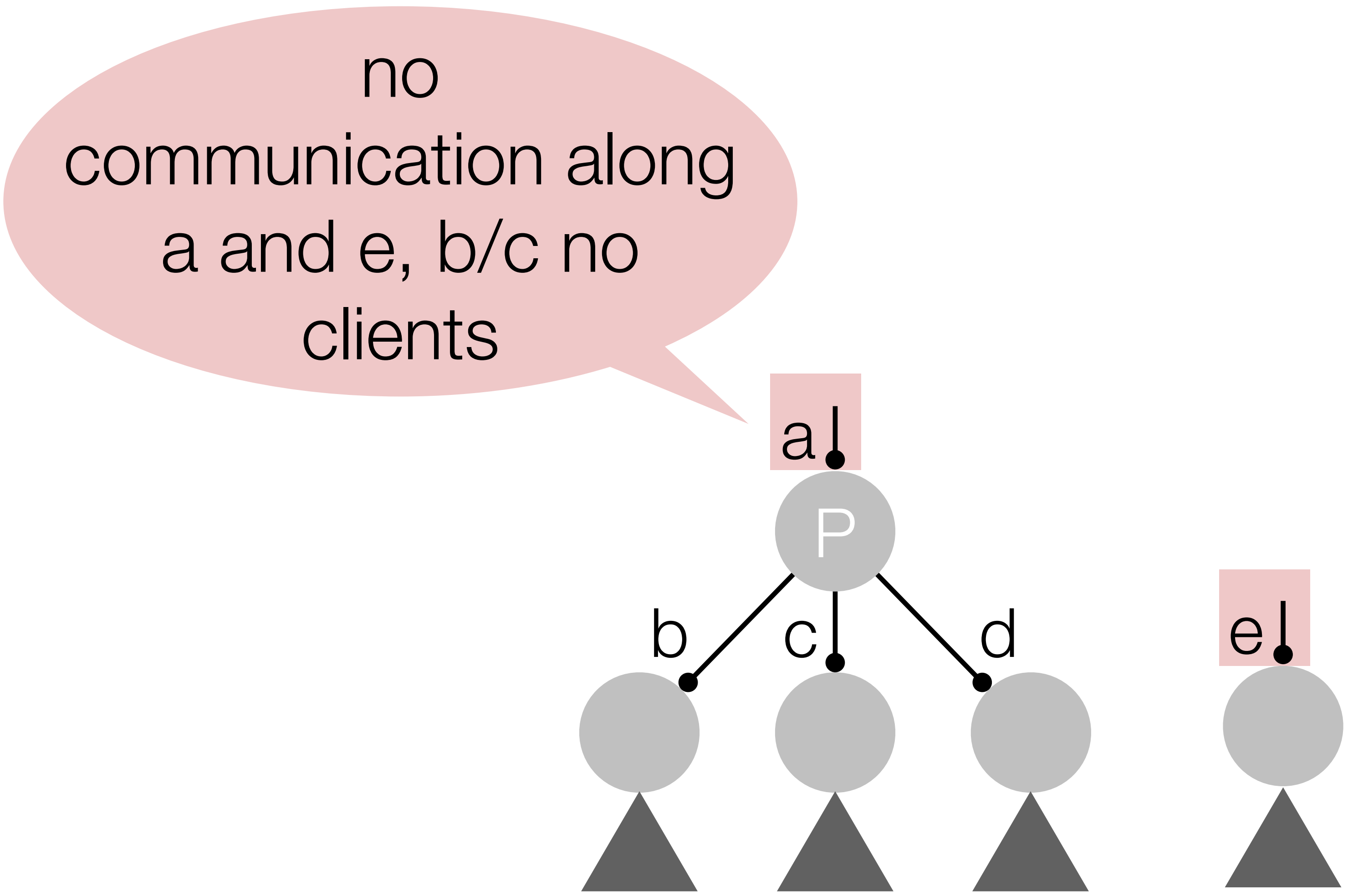
Preservation and progress

Theorem (Preservation). *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.



Preservation and progress

Theorem (Preservation). *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.



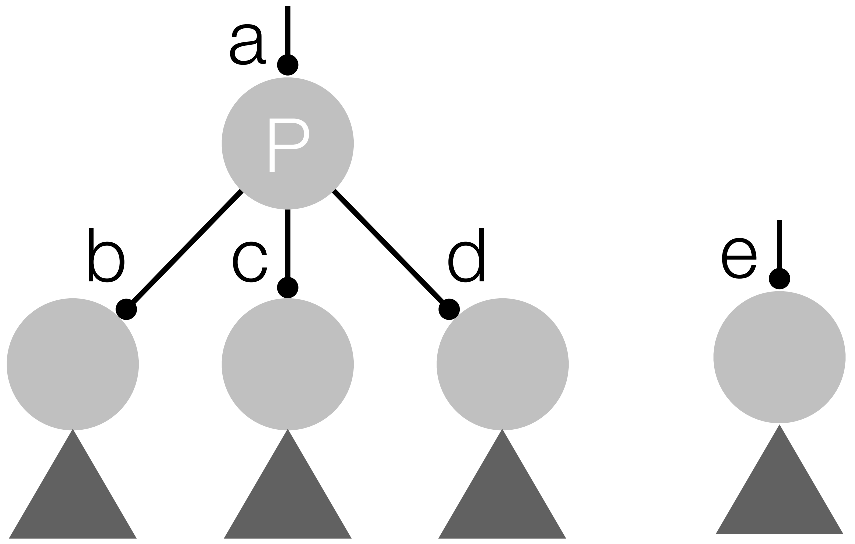
Preservation and progress

Theorem (Preservation)**.** *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.

Theorem (Progress)**.** *If* $\vDash \Omega :: \Delta$, *then either*

1. $\Omega \longrightarrow \Omega'$, *for some* Ω', *or*

2. Ω *is poised.*



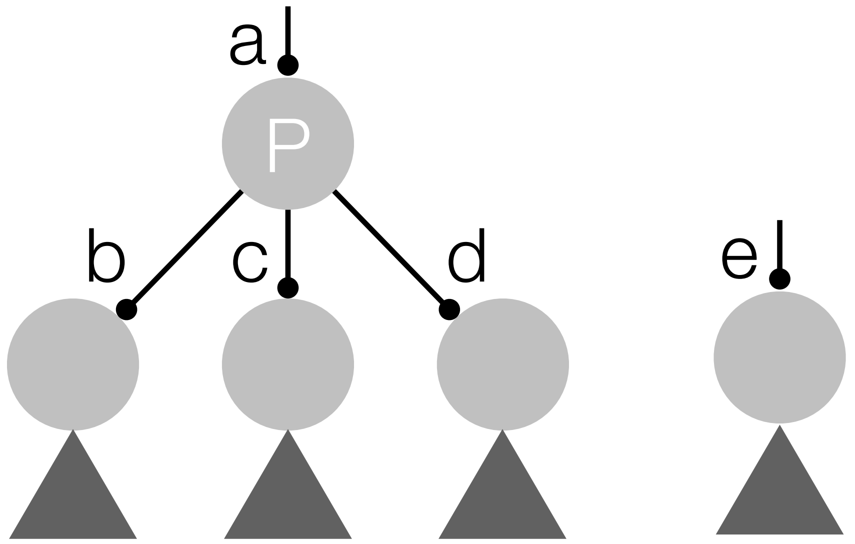
Preservation and progress

Theorem (Preservation). *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.

Theorem (Progress). *If* $\vDash \Omega :: \Delta$, *then either*

1. $\Omega \longrightarrow \Omega'$, *for some* Ω', *or*

2. Ω *is poised.*



Preservation and progress

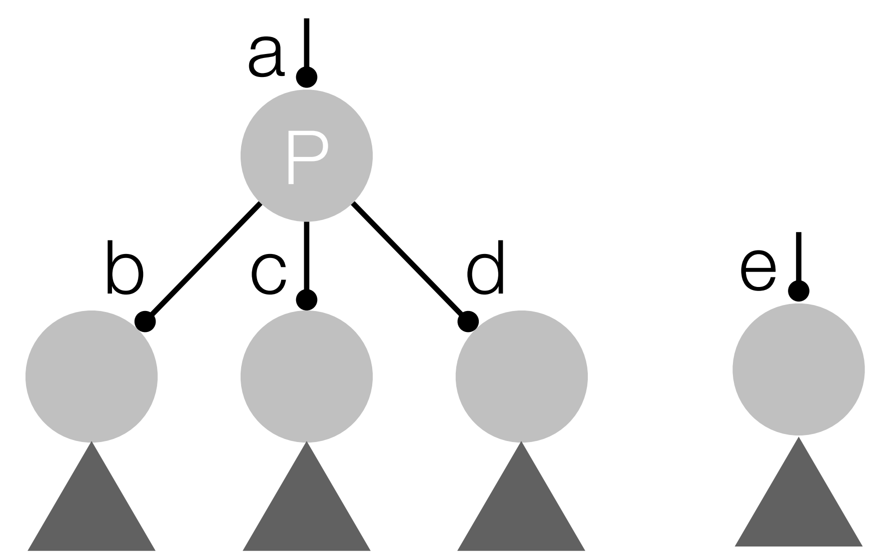
Theorem (Preservation). *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.

Theorem (Progress). *If* $\vDash \Omega :: \Delta$, *then either*

1. $\Omega \longrightarrow \Omega'$, *for some* Ω', *or*

2. Ω *is poised.*

every
process poised



Preservation and progress

Theorem (Preservation). *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.

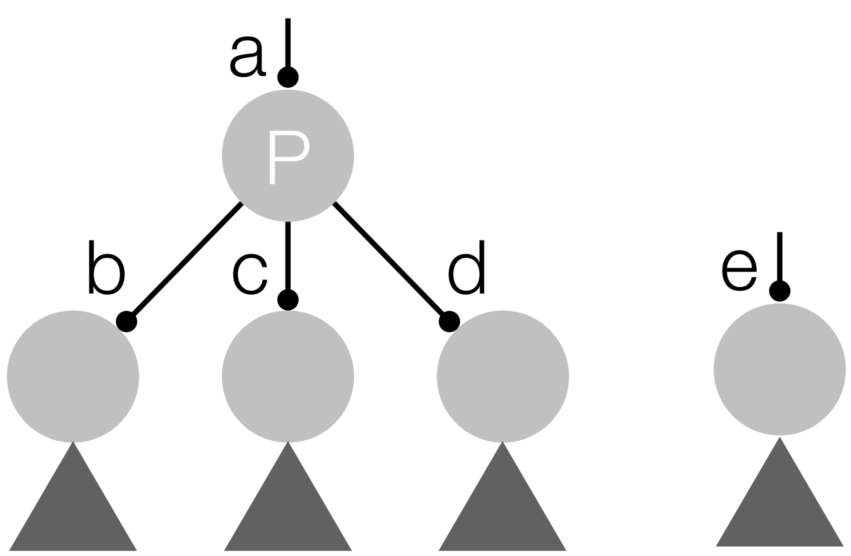
Theorem (Progress). *If* $\vDash \Omega :: \Delta$, *then either*

1. $\Omega \longrightarrow \Omega'$, *for some* Ω', *or*

2. Ω *is poised.*

every process poised

a poised process is ready to sync along offering channel



Preservation and progress

Theorem (Preservation). *If* $\vDash \Omega :: \Delta$ *and* $\Omega \longrightarrow \Omega'$, *then* $\vDash \Omega' :: \Delta$.

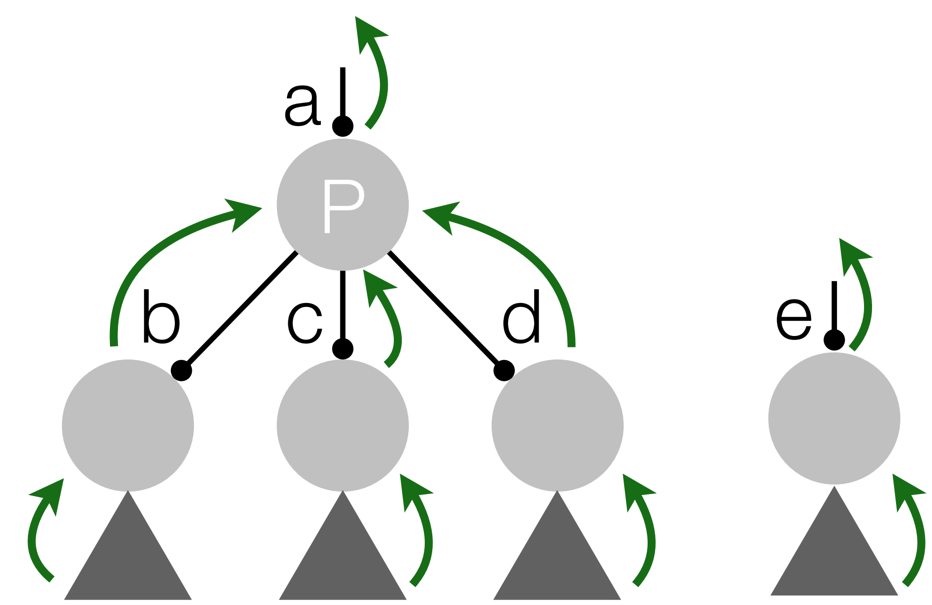
Theorem (Progress). *If* $\vDash \Omega :: \Delta$, *then either*

1. $\Omega \longrightarrow \Omega'$, *for some* Ω', *or*

2. Ω *is poised.*

every process poised

a poised process is ready to sync along offering channel



Taking stock

Taking stock

➡ linear session type language

Taking stock

- linear session type language
  - guarantees session fidelity and deadlock-freedom

Taking stock

- linear session type language
  - guarantees session fidelity and deadlock-freedom
  - corresponds to intuitionistic linear logic

Taking stock

- linear session type language
  - guarantees session fidelity and deadlock-freedom
  - corresponds to intuitionistic linear logic
- one connective from linear logic still missing: persistent truth

Taking stock

- linear session type language
  - guarantees session fidelity and deadlock-freedom
  - corresponds to intuitionistic linear logic
- one connective from linear logic still missing: persistent truth

A, B	$\triangleq$	$A \otimes B$	multiplicative conjunction	"channel output"
		$A \multimap B$	multiplicative implication	"channel input"
		$A \mathbin{\&} B$	additive conjunction	"external choice"
		$A \oplus B$	additive disjunction	"internal choice"
		$\mathbf{1}$	unit for $\otimes$	"termination"

Taking stock

- linear session type language
  - guarantees session fidelity and deadlock-freedom
  - corresponds to intuitionistic linear logic
- one connective from linear logic still missing: persistent truth

A, B	$\triangleq$	$A \otimes B$	multiplicative conjunction	"channel output"
		$A \multimap B$	multiplicative implication	"channel input"
		$A \,\&\, B$	additive conjunction	"external choice"
		$A \oplus B$	additive disjunction	"internal choice"
		$\mathbf{1}$	unit for $\otimes$	"termination"
		$!A$	"of course", persistent truth	"replication"