Nominal game semantics and automata

Nikos Tzevelekos

Queen Mary University of London

MSR, Cambridge, Dec 2014

Supported by a Royal Academy of Engineering Research Fellowship

Computation with names (Java)

```
...
  public void foo() {
    // Create new list
    List x = new ArrayList();

    x.add(1); x.add(2);
    Iterator i = x.iterator();
    Iterator j = x.iterator();
    i.next(); i.remove(); j.next();
  }
...
```

new creates a fresh name:

- different from any previous one
- comparable for equality

Computation with names (ML)

type with only value: ()

“pure” names

$$\lambda x.\ \mathrm{ref}() \ :\ \mathrm{unit} \rightarrow (\mathrm{unit\ ref})$$

Computation with names (ML)

type with only value: ()

"pure" names

$$\lambda x.\ \mathrm{ref}()\ :\ \mathrm{unit} \rightarrow (\mathrm{unit\ ref})$$

$$\mathrm{let}\ f = [\ \text{-}\ ]\ \mathrm{in}\ f\,() == f\,()$$

Computation with names (ML)

type with only value: ()

“pure” names

$$\lambda x.\ \mathrm{ref}()\ :\ \mathrm{unit} \rightarrow (\mathrm{unit\ ref})$$

$$\mathrm{let}\ f = [\ \text{-}\ ]\ \mathrm{in}\ \ f\,() == f\,()$$

Computation with names (ML)

type with only value: ()

“pure” names

$$\lambda x.\ \mathrm{ref}()\ :\ \mathrm{unit} \longrightarrow (\mathrm{unit\ ref})$$

$$\mathrm{let}\ f = [\ \text{-}\ ]\ \mathrm{in}\ \ f\,() == f\,()$$

$$(\lambda x.\ \mathrm{ref}())\ () ==\ (\lambda x.\ \mathrm{ref}())\ ()$$

Computation with names (ML)

type with only value: ()

“pure” names

$\lambda x.\ \mathrm{ref}()\ :\ \mathrm{unit} \longrightarrow (\mathrm{unit\ ref})$

$\mathrm{let}\ f = [\ \text{-}\ ]\ \mathrm{in}\ f\,() == f\,()$

$(\lambda x.\ \mathrm{ref}())\ () == (\lambda x.\ \mathrm{ref}())\ ()$

false

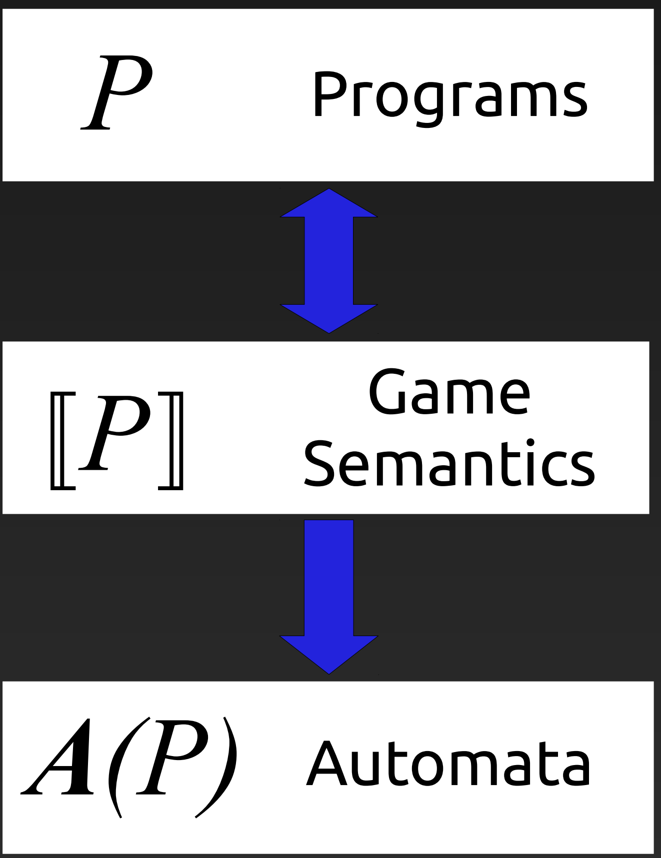
What this talk is about

We give an overview of techniques for modelling and analysing computation with names

Names are an abstraction for handling dynamic resources: *references, objects, channels, exceptions*…

Our approach starts denotationally, using game semantics, and reaches concrete algorithms using automata over infinite alphabets

Overview



Example

Call-by-value language with unit references

$$\lambda x.\, \mathrm{ref}() \;\ncong\; \mathrm{let}\ y = \mathrm{ref}()\ \mathrm{in}\ \lambda x.y \;:\; \mathrm{unit} \to \mathrm{unit\ ref}$$

$$P \;\cong\; P'$$

same observable behaviour in every context

Example

Call-by-value language with unit references

$$\lambda x.\ \mathrm{ref}() \ \ncong\ \mathrm{let}\ y = \mathrm{ref}()\ \mathrm{in}\ \lambda x.y \ :\ \mathrm{unit} \to \mathrm{unit\ ref}$$

$$\mathrm{let}\ y = \mathrm{ref}()\ \mathrm{in}\ \lambda x.(x == y) \ \cong\ \lambda x.\ \mathrm{false} \ :\ \mathrm{unit} \to \mathrm{bool}$$

$$P \cong P'$$

same observable behaviour in every context

Example

Call-by-value language with unit references

"Create a *fresh reference*; then *compare* input references with the fresh one and return the result of the comparison"

$$\lambda x.\ \mathrm{ref}() \ \not\cong \ \mathrm{let}\ y = \mathrm{ref}()\ \mathrm{in}\ \lambda x.y \ : \ \mathrm{unit} \longrightarrow \mathrm{unit\ ref}$$

$$\mathrm{let}\ y = \mathrm{ref}()\ \mathrm{in}\ \lambda x.(x == y) \ \cong \ \lambda x.\ \mathrm{false} \ : \ \mathrm{unit} \longrightarrow \mathrm{bool}$$

$$P \ \cong \ P'$$

same observable behaviour in every context

Quiz

Call-by-value language with int references

$$g : \text{int ref} \rightarrow \text{unit} \;\;\vdash\;\; \text{let } x,y = \text{ref}(0) \text{ in } (g\,x);\, y := !x;\, y \;\;:\;\; \text{int ref}$$

vs

$$g : \text{int ref} \rightarrow \text{unit} \;\;\vdash\;\; \text{let } x = \text{ref}(0) \text{ in } (g\,x);\, x \;\;:\;\; \text{int ref}$$

Can we capture real "meaning"?

How to assign denotations to programs,

$$[\![\_]\!] : \text{Syntax} \longrightarrow \mathcal{M}$$

such that:

$$P \cong P' \Leftrightarrow [\![P]\!] = [\![P']\!]$$

Can we capture real "meaning"?

How to assign denotations to programs,

$$[\![\_]\!] : \text{Syntax} \longrightarrow \mathcal{M}$$

such that:

$$P \cong P' \Leftrightarrow [\![P]\!] = [\![P']\!]$$

full abstraction

Can we capture real "meaning"?

How to assign denotations to programs,

$$[\![\_]\!] : \text{Syntax} \longrightarrow \mathcal{M}$$

such that:

$$P \cong P' \Leftrightarrow [\![P]\!] = [\![P']\!]$$

Here the Syntax will be some fragment of ML/F#,
i.e. call-by-value PCF + references + exceptions + …

$$T \ ::= \ \text{unit} \mid \text{int} \mid T\,\text{ref} \mid T \rightarrow T \mid \ldots$$

$$M == M \mid \ldots$$

$$M ::= \ () \mid i \mid a \mid x \mid \lambda x.M \mid M\,M \mid \text{ref}\,M \mid M := M \mid\, !M \mid$$

Game Semantics

Game Semantics

- Computation is modelled as a 2-player game between:
  - *Opponent* (the environment), aka O
  - *Proponent* (the program), aka P
- Qualitative games (≠ Game Theory)
- Programs = *strategies* for P
- *Categories* of games

Games played in arenas

$$x_1 : T_1,\ \ldots,\ x_n : T_n \vdash M : T$$

Games played in arenas

program

free variables

output type

$$x_1 : T_1,\ \ldots,\ x_n : T_n \vdash M : T$$

input types

Games played in arenas

free variables

program

output type

$$x_1 : T_1,\ \ldots,\ x_n : T_n \vdash M : T$$

input types

$$[\![M]\!] : [\![T_1,\ \ldots,\ T_n]\!] \longrightarrow [\![T]\!]$$

Games played in arenas

free variables

program

output type

$$x_1 : T_1, \ \ldots, \ x_n : T_n \vdash M : T$$

input types

$$[\![M]\!] \ : \ [\![T_1, \ \ldots, \ T_n]\!] \longrightarrow [\![T]\!]$$

strategy

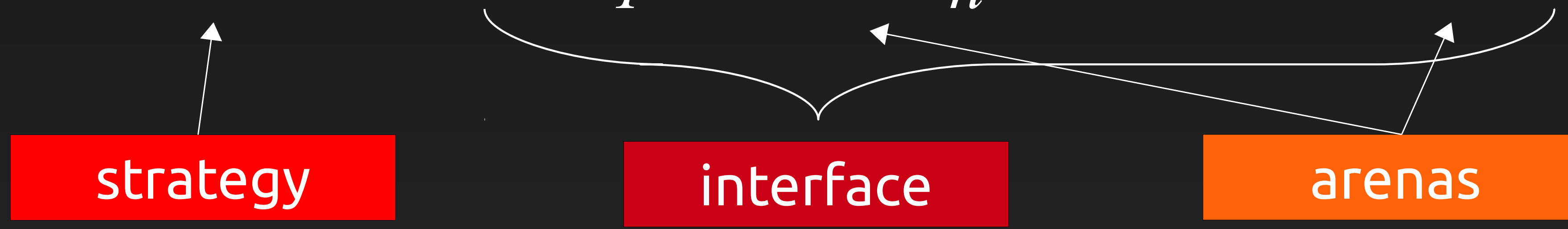
interface

(aka prearena)

arenas

Arenas of moves

$$[\![M]\!] : [\![T_1, \ldots, T_n]\!] \longrightarrow [\![T]\!]$$



Arenas of moves

$$[\![M]\!] : [\![T_1, \ldots, T_n]\!] \longrightarrow [\![T]\!]$$

arenas

moves

$$[\![\text{unit}]\!] = \{ * \}$$

$$[\![\text{int}]\!] = \{ 0, 1, \text{-}1, \ldots \}$$

$$[\![T\,\text{ref}]\!] = \{ a, b, \ldots \} \qquad a, b, \ldots \in \mathcal{N}_T$$

...

Arenas of moves

$$[\![M]\!] : [\![T_1, \ldots, T_n]\!] \longrightarrow [\![T]\!]$$

arenas

$\mathcal{N}_T$ a set of *names*:

- infinitely many
- comparable for equality only

$$[\![\mathrm{unit}]\!] = \{ * \}$$

$$[\![\mathrm{int}]\!] = \{ 0, 1, \text{-}1, \ldots \}$$

$$[\![T\,\mathrm{ref}]\!] = \{ a, b, \ldots \}$$

$$a, b, \ldots \in \mathcal{N}_T$$

Arenas of moves

$$[\![M]\!] : [\![T_1, \ldots, T_n]\!] \longrightarrow [\![T]\!]$$

arenas

$\mathcal{N}_T$ a set of *names*:

- infinitely many
- comparable for equality only

$$[\![\text{unit}]\!] = \{ * \} = \mathbf{1}$$

$$[\![\text{int}]\!] = \{ 0, 1, -1, \ldots \} = \mathbb{Z}$$

$$[\![T\,\text{ref}]\!] = \{ a, b, \ldots \} = \mathbb{A}_T \qquad a, b, \ldots \in \mathcal{N}_T$$

A simple interface

$$\mathbb{Z} \longrightarrow \mathbb{Z}$$

(i.e. calls)

questions

$\{\ 0,\ 1,\ -1,\ \ldots\ \}$

(i.e. returns)

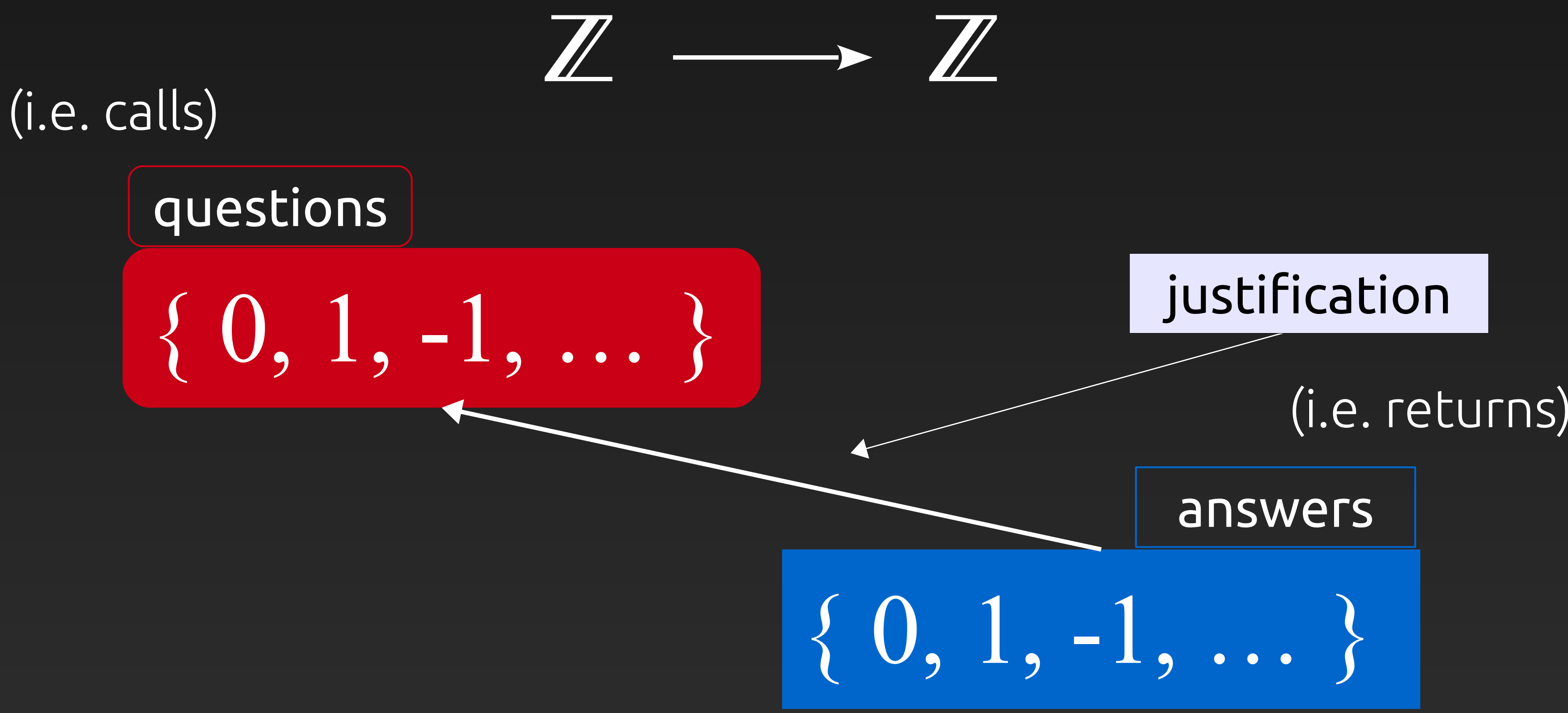
answers

$\{\ 0,\ 1,\ -1,\ \ldots\ \}$

O : Opponent

P : Proponent

A simple interface



O : Opponent

P : Proponent

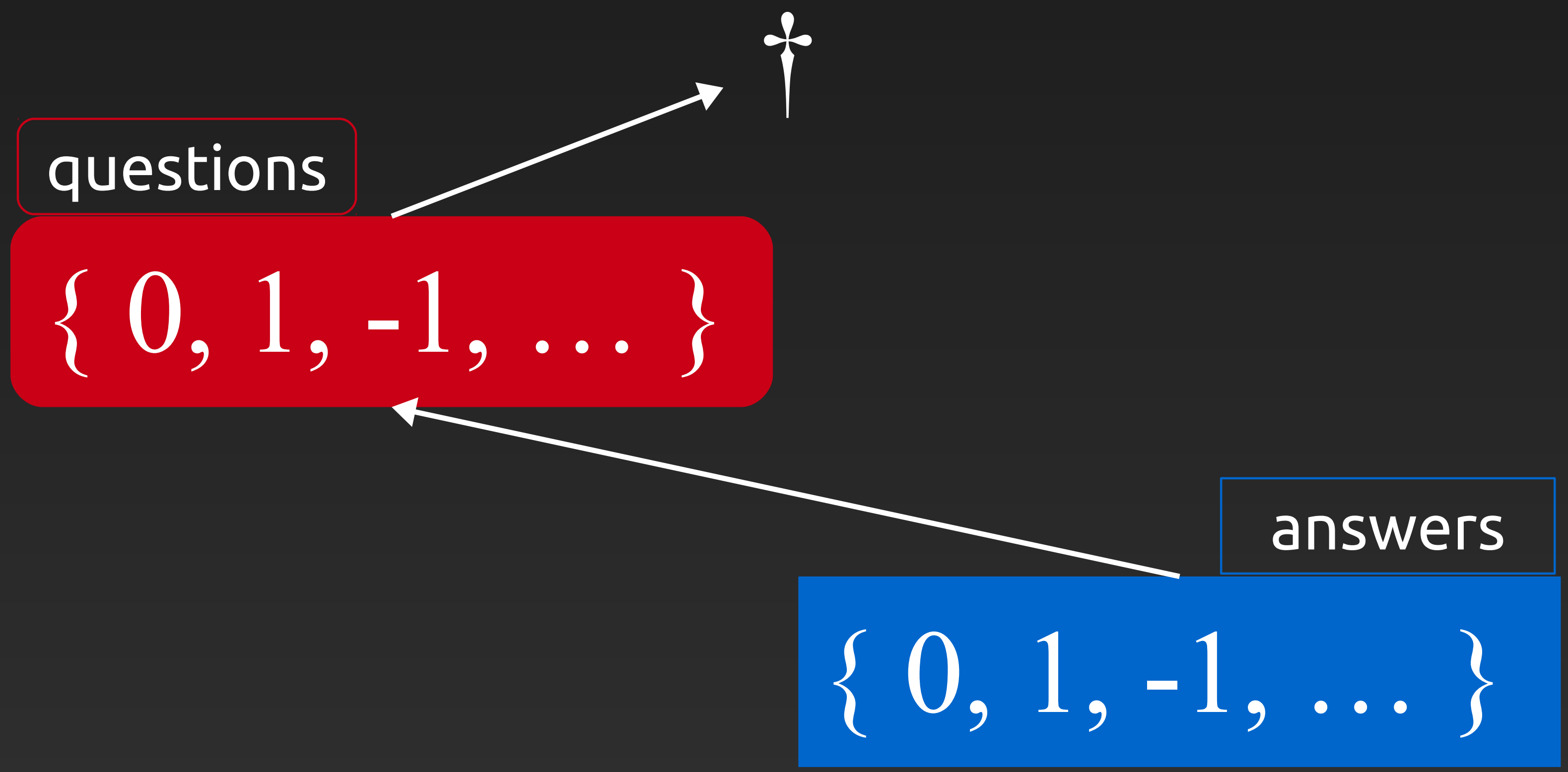
A higher-order arena

$$[\![\mathrm{int} \longrightarrow \mathrm{int}]\!]$$

$\dagger$ ← *"here is a function"*

A higher-order arena

$$[\![\mathrm{int} \longrightarrow \mathrm{int}]\!]$$

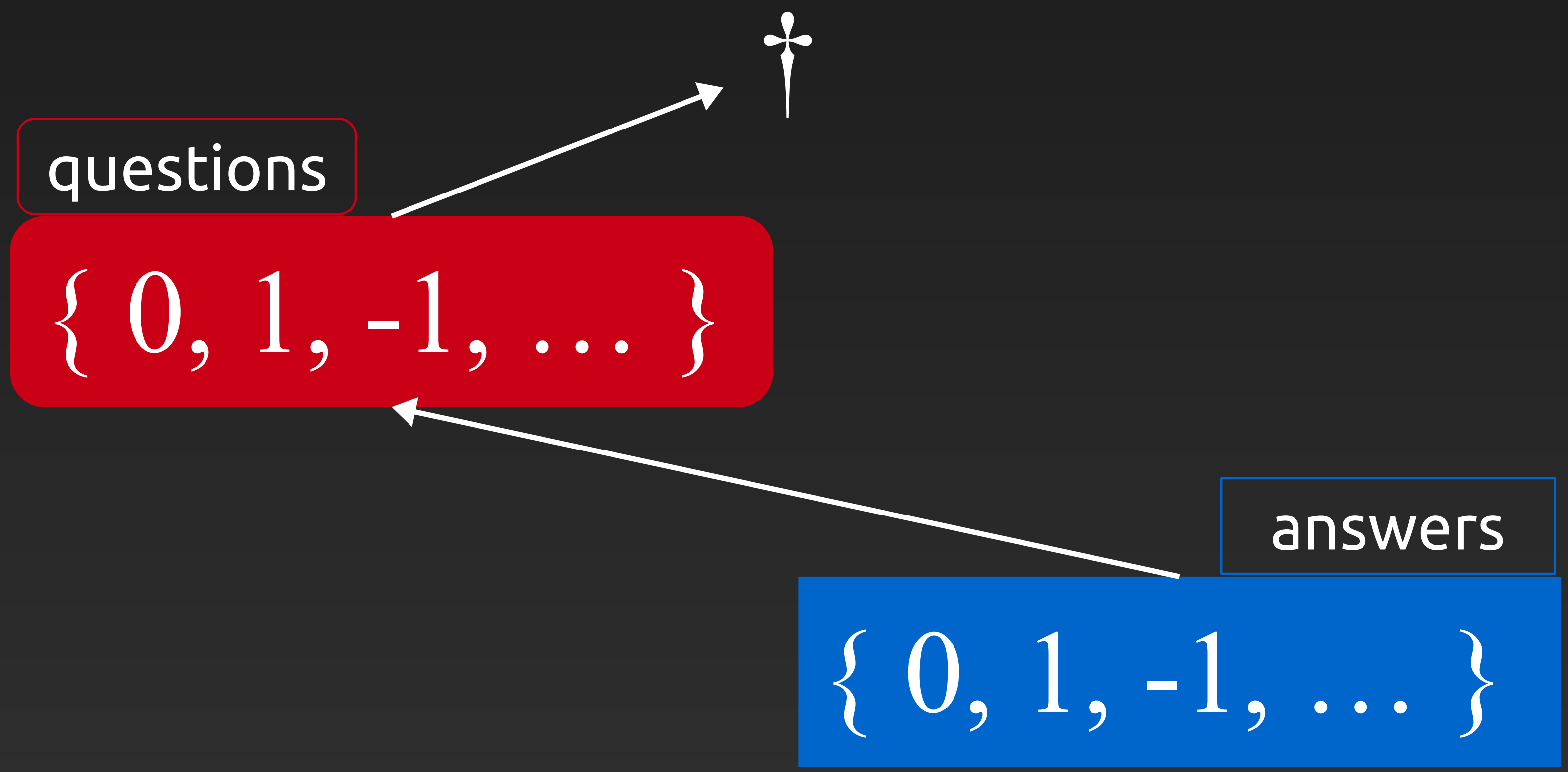


O : Opponent

P : Proponent

A higher-order arena

$$[\![\mathrm{int} \longrightarrow \mathrm{int}]\!] = \mathbb{Z} \Rightarrow \mathbb{Z}$$

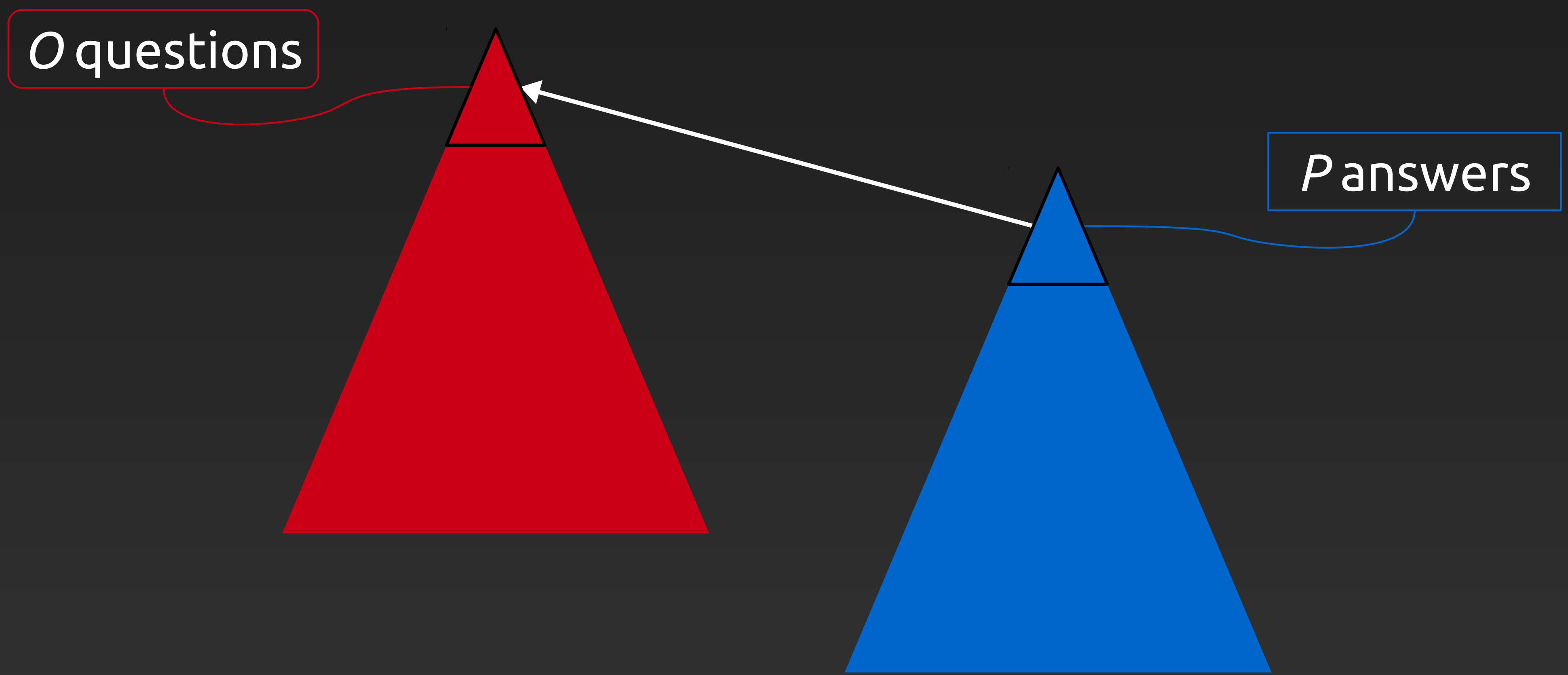


O : Opponent

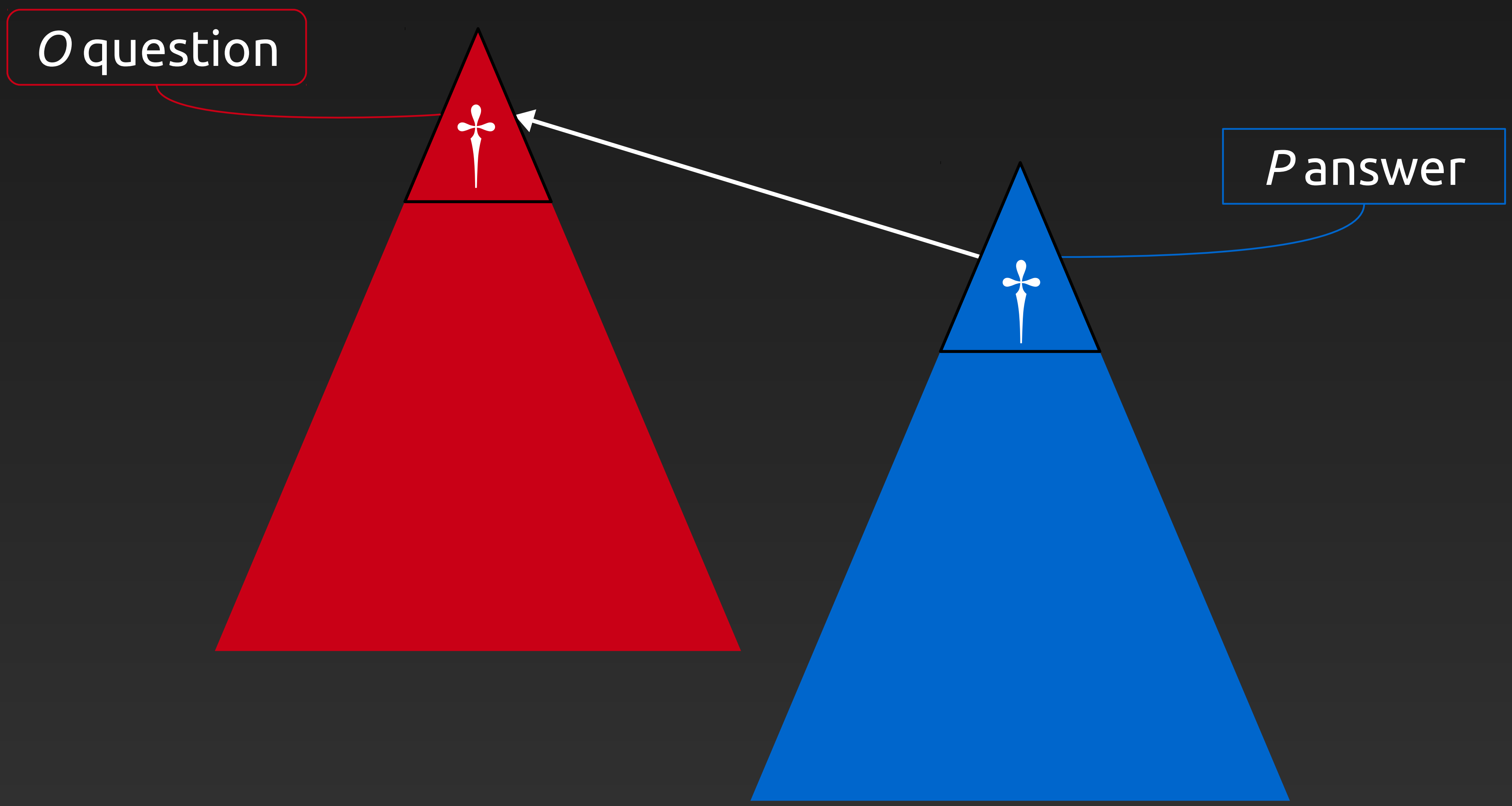
P : Proponent

A higher-order interface

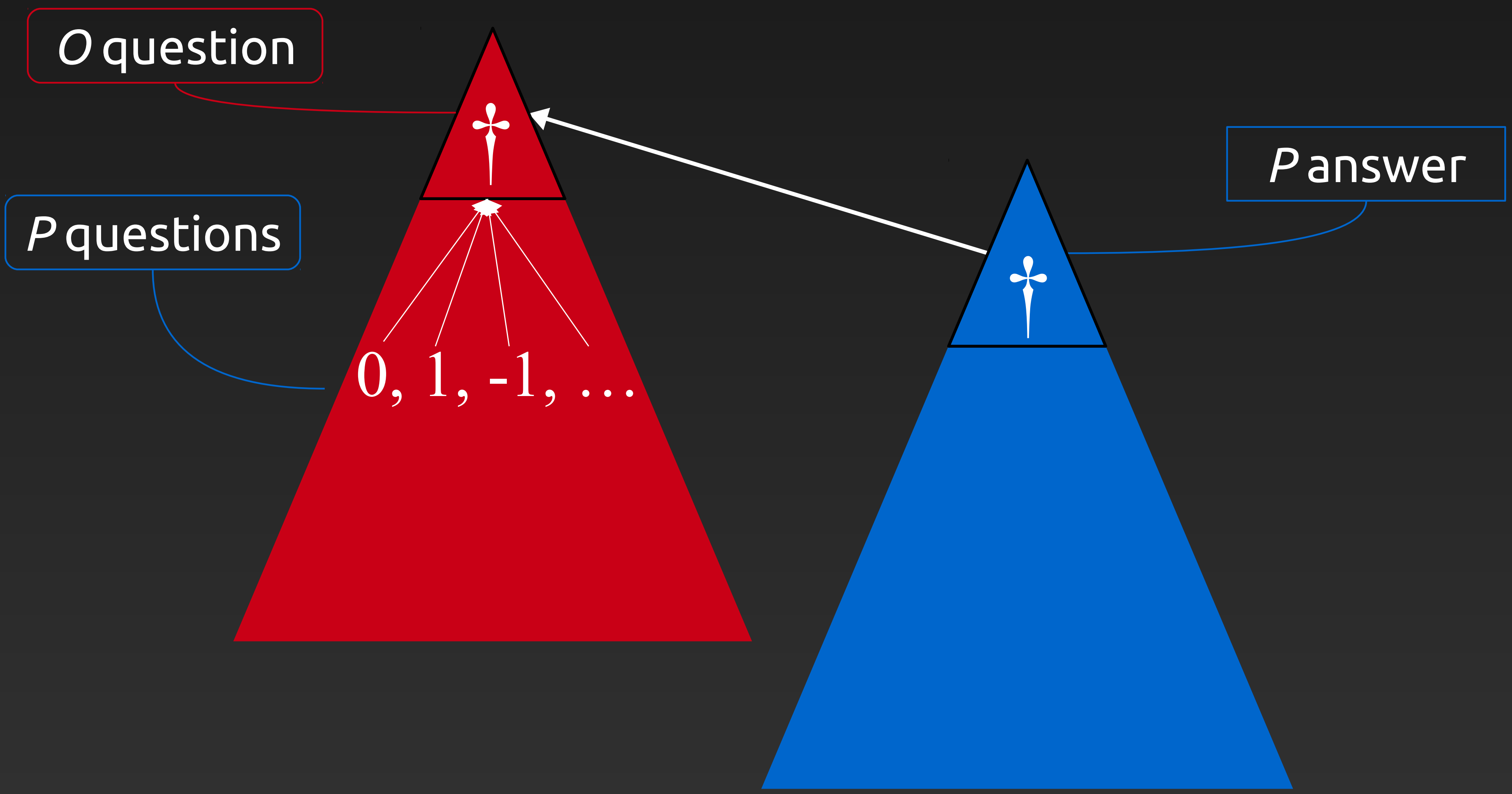
$$[\![T]\!] \longrightarrow [\![T']\!]$$

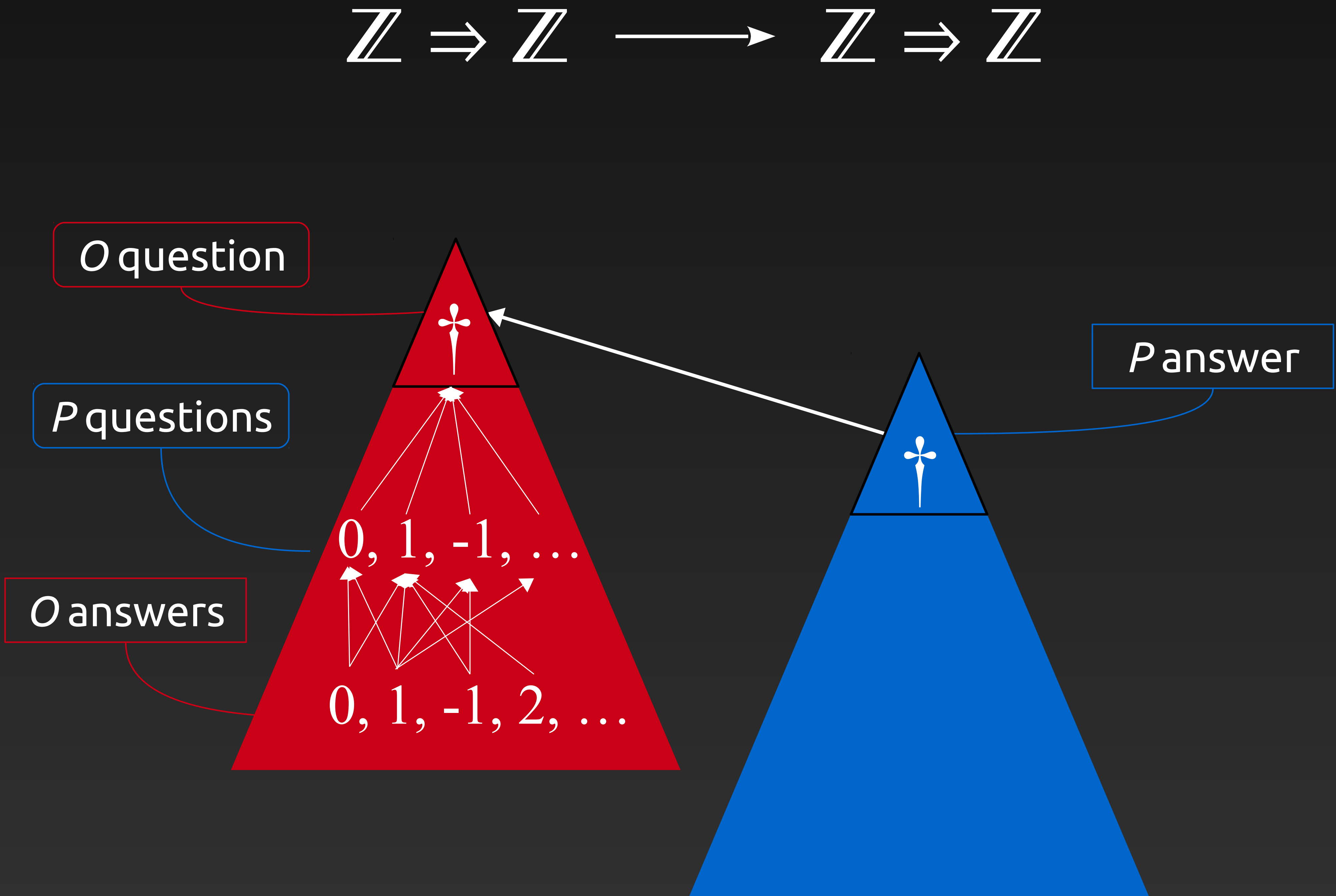


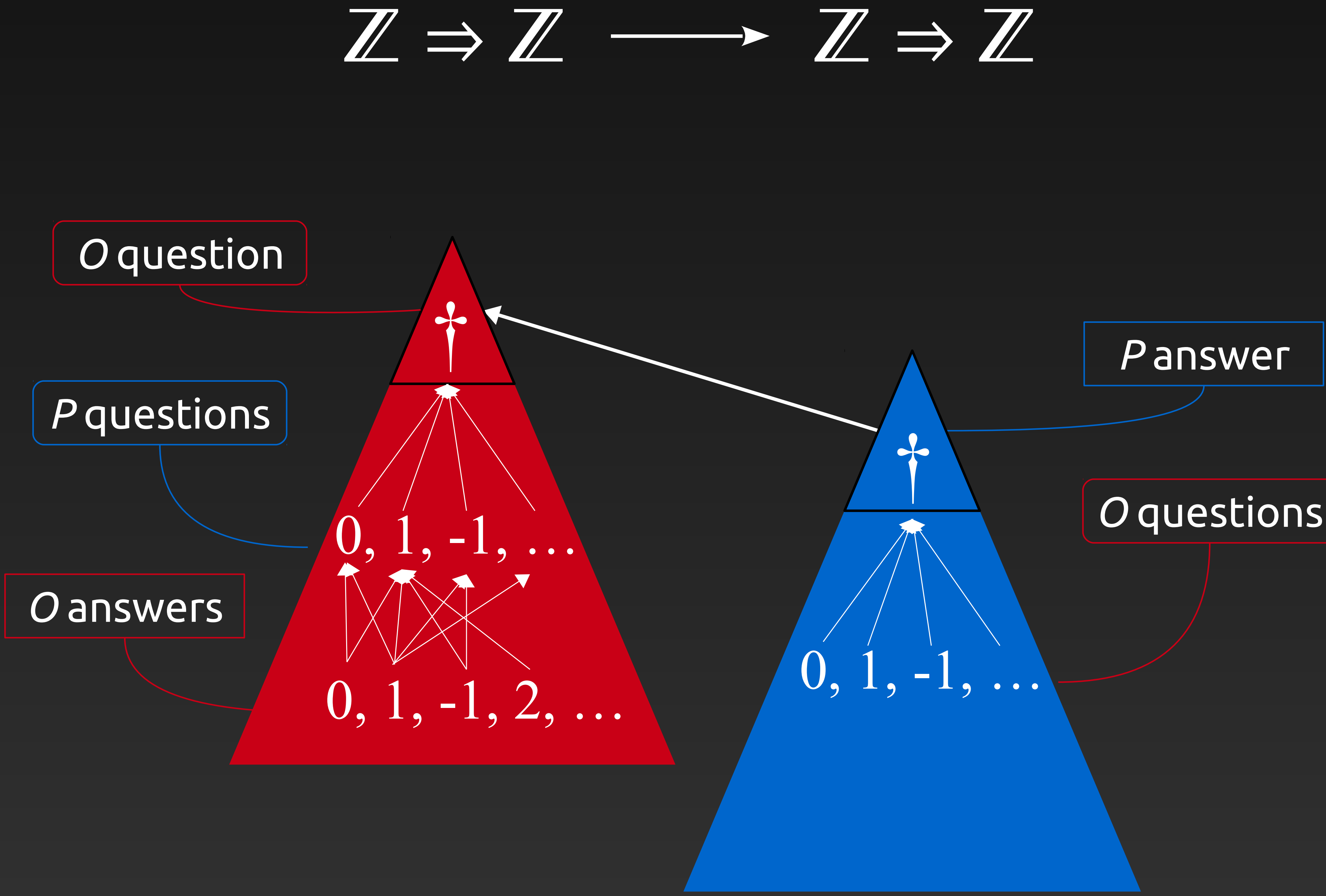
$$\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}$$

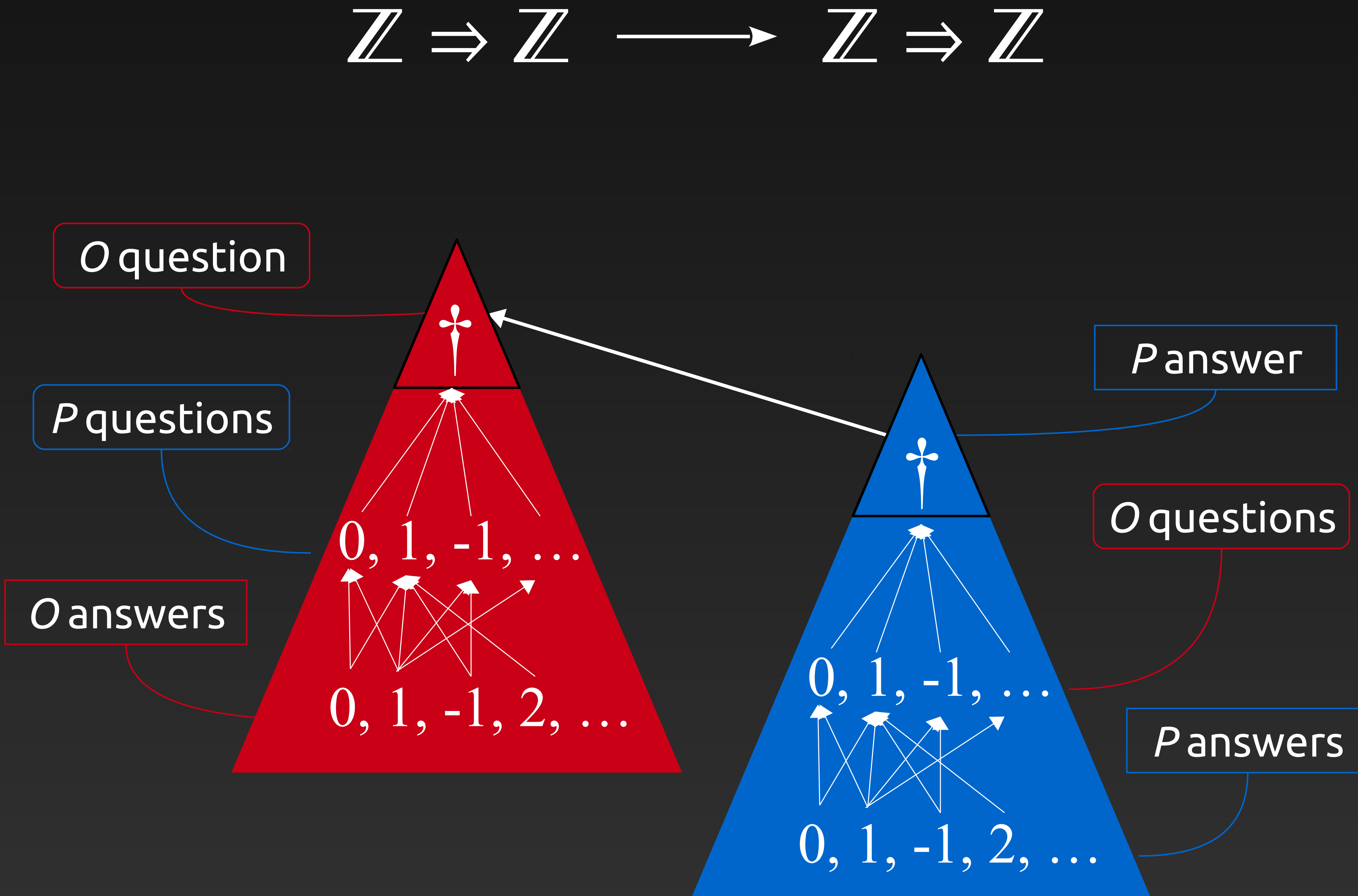


$$\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}$$



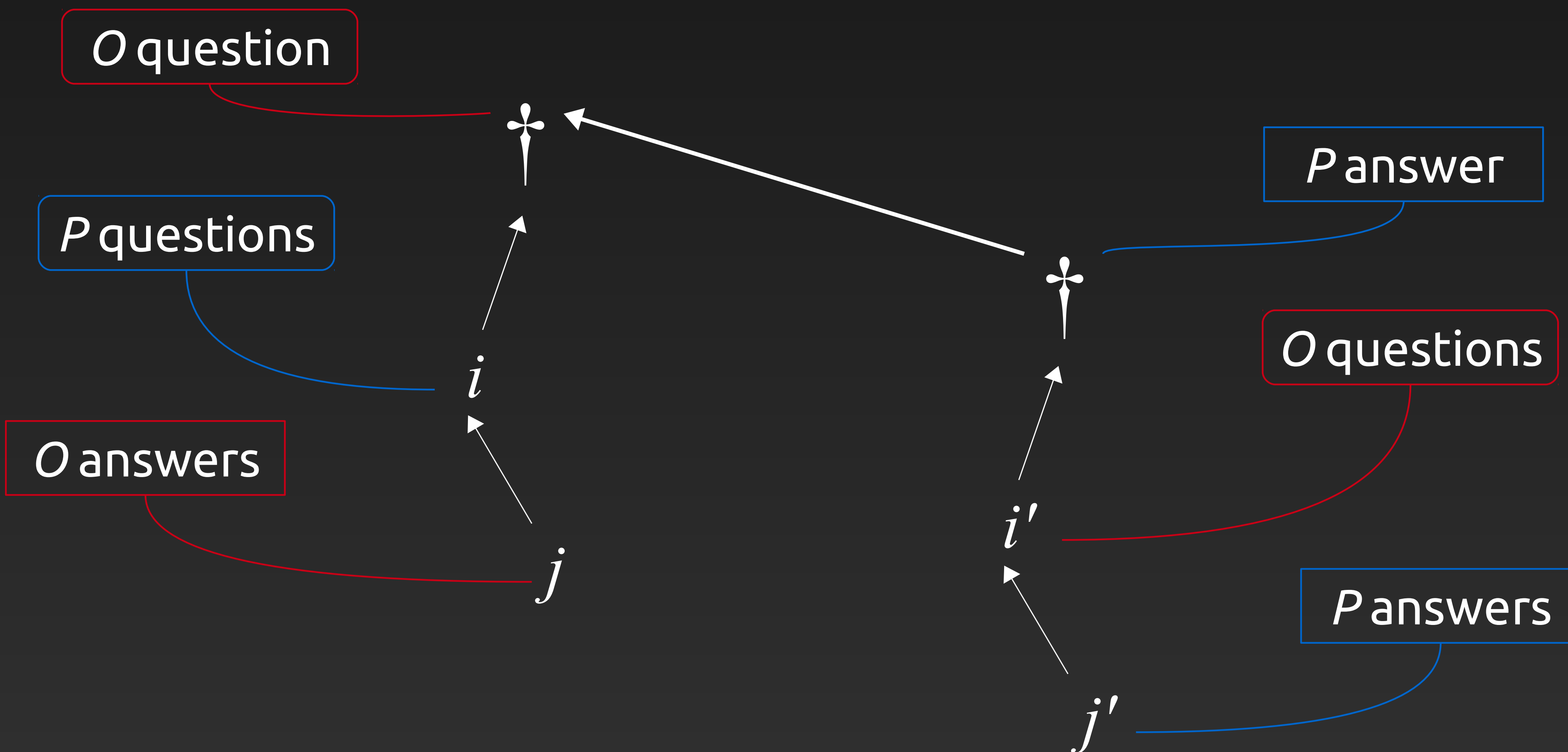

ℤ ⇒ ℤ ⟶ ℤ ⇒ ℤ
O question
P questions
O answers
P answer
0, 1, -1, …
0, 1, -1, 2, …


$\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}$
O question
P questions
O answers
0, 1, -1, …
0, 1, -1, 2, …
P answer
O questions
0, 1, -1, …



$\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}$
O question
P questions
0, 1, -1, . . .
O answers
0, 1, -1, 2, . . .
P answer
O questions
0, 1, -1, . . .
P answers
0, 1, -1, 2, . . .

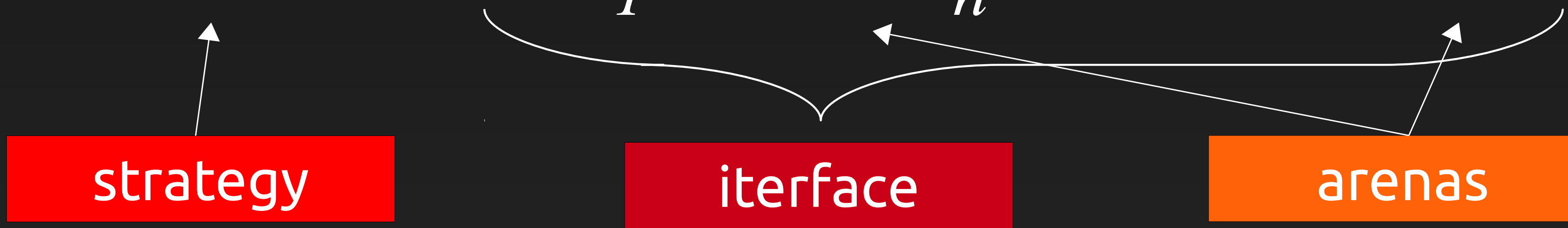
$$\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}$$



$$i, j, i', j' = 0,\ 1,\ \text{-}1,\ 2,\ \text{-}2,\ \ldots$$

Strategies

$$[\![M]\!] : [\![T_1, \ldots, T_n]\!] \longrightarrow [\![T]\!]$$



Strategies are “instructions” for *P* on how to play on a given interface

- Formally, sets of even-length plays satisfying combinatorial conditions linked to language expressivity

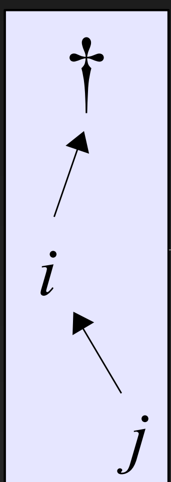
Example

$$f : \text{int} \to \text{int} \;\vdash\; \lambda x. f\, x + 1 : \text{int} \to \text{int}$$

$$\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}$$

Example

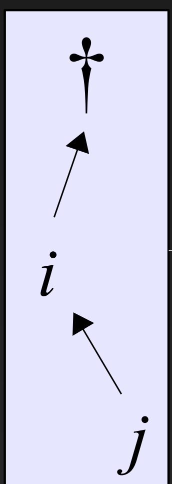
$$f : \mathrm{int} \to \mathrm{int} \;\vdash\; \lambda x.\, f\, x + 1 : \mathrm{int} \to \mathrm{int}$$



$$\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}$$

Example

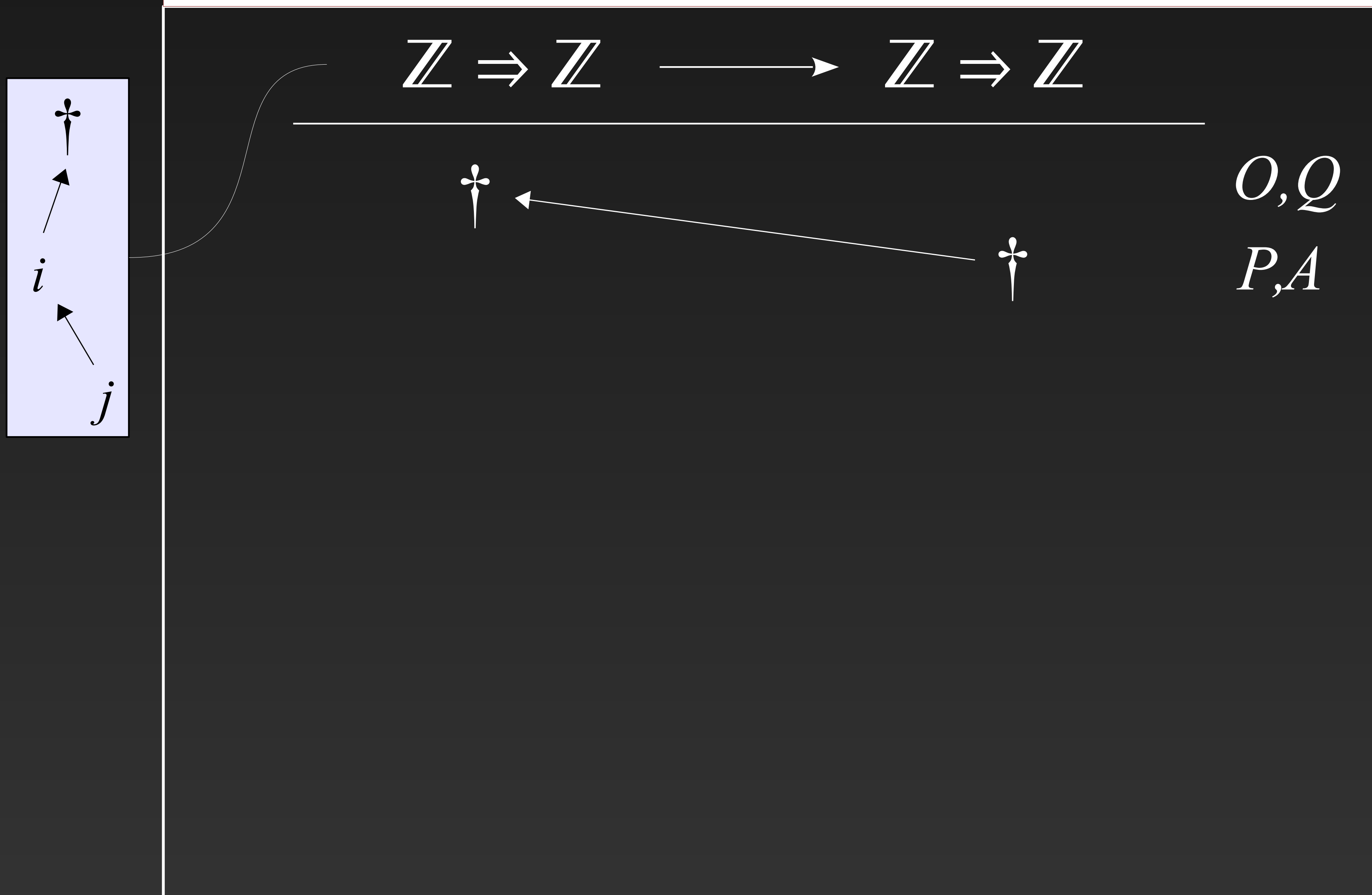
$$f : \text{int} \rightarrow \text{int} \;\vdash\; \lambda x.\, f\, x + 1 : \text{int} \rightarrow \text{int}$$



$$\frac{\mathbb{Z} \Rightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \Rightarrow \mathbb{Z}}{\dagger} \; O,Q$$

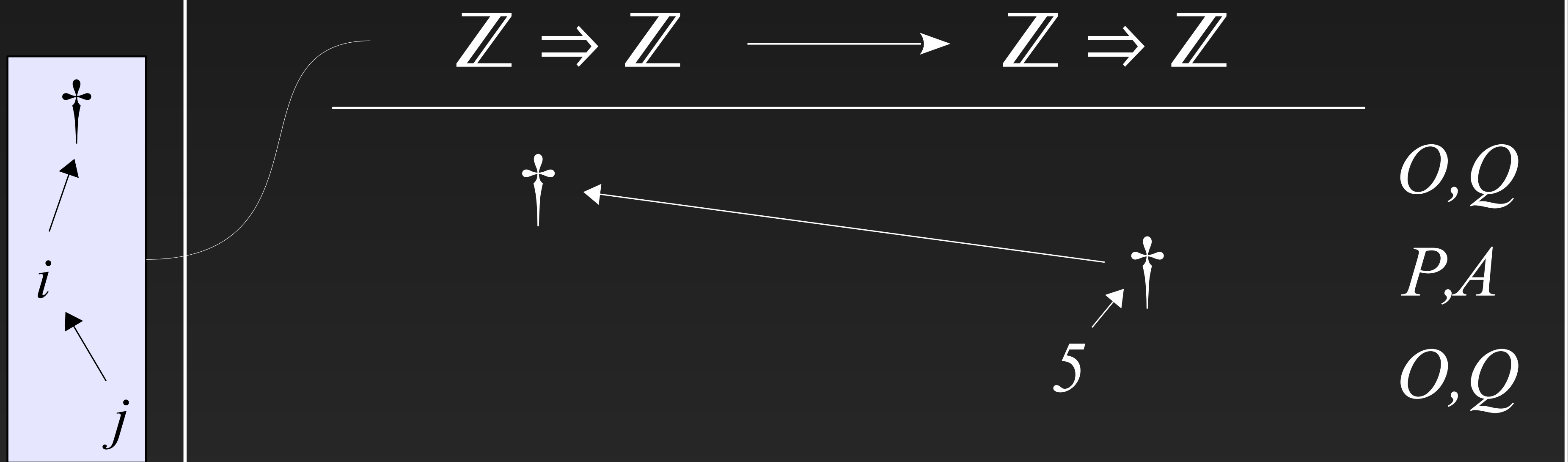
Example

$$f : \text{int} \to \text{int} \;\vdash\; \lambda x.\, f\,x + 1 : \text{int} \to \text{int}$$



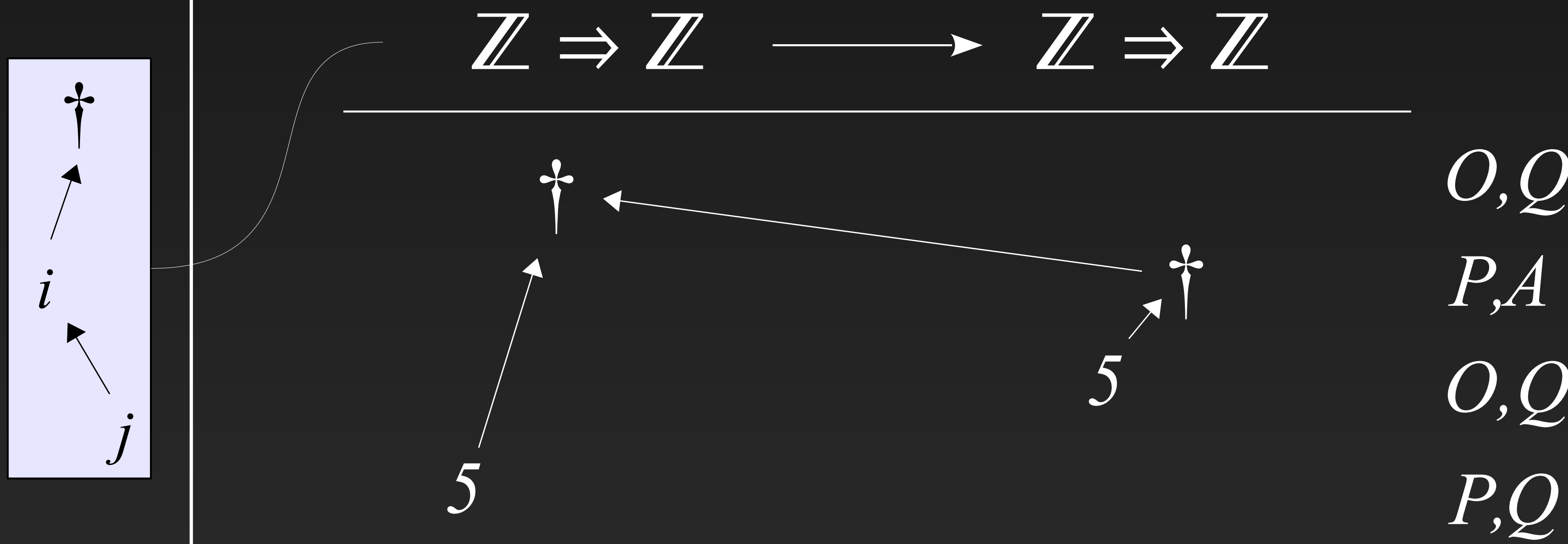
Example

$$f : \mathrm{int} \to \mathrm{int} \;\vdash\; \lambda x.\, f\,x + 1 : \mathrm{int} \to \mathrm{int}$$



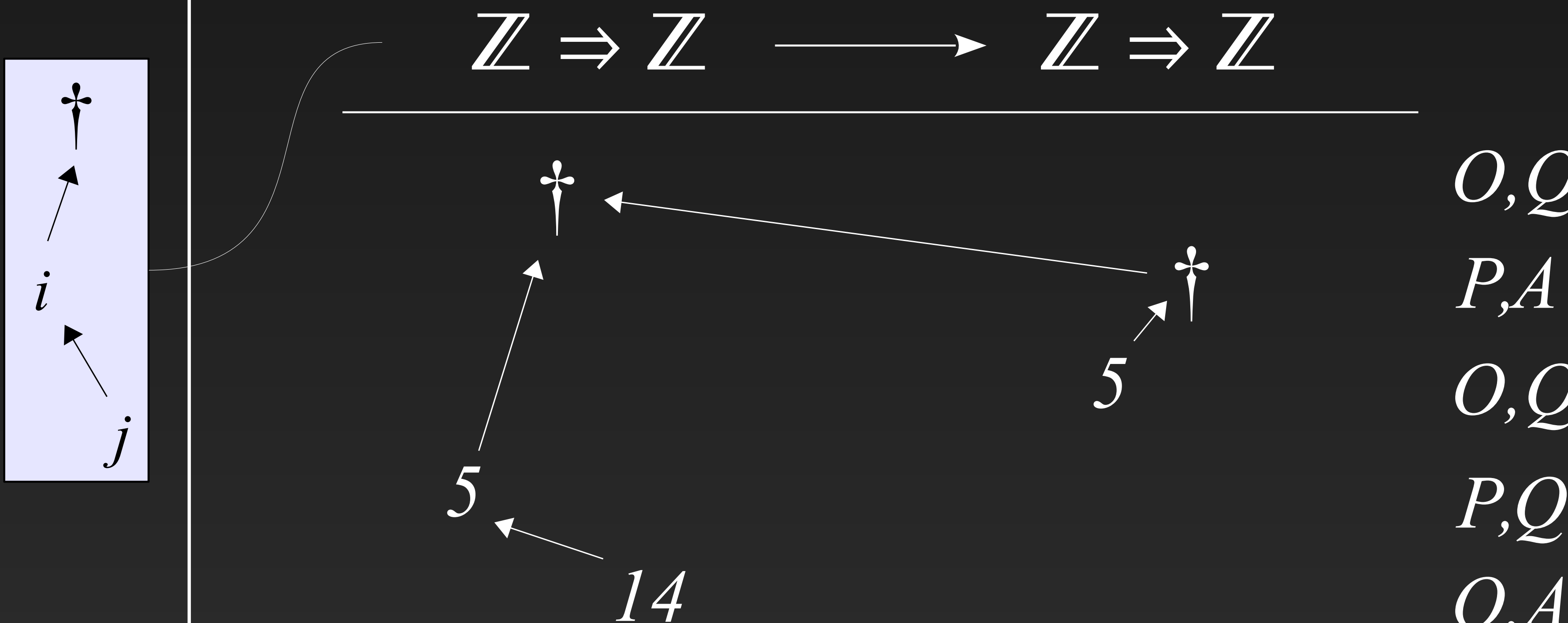
Example

$$f : \mathrm{int} \to \mathrm{int} \;\vdash\; \lambda x.\, f\,x + 1 : \mathrm{int} \to \mathrm{int}$$



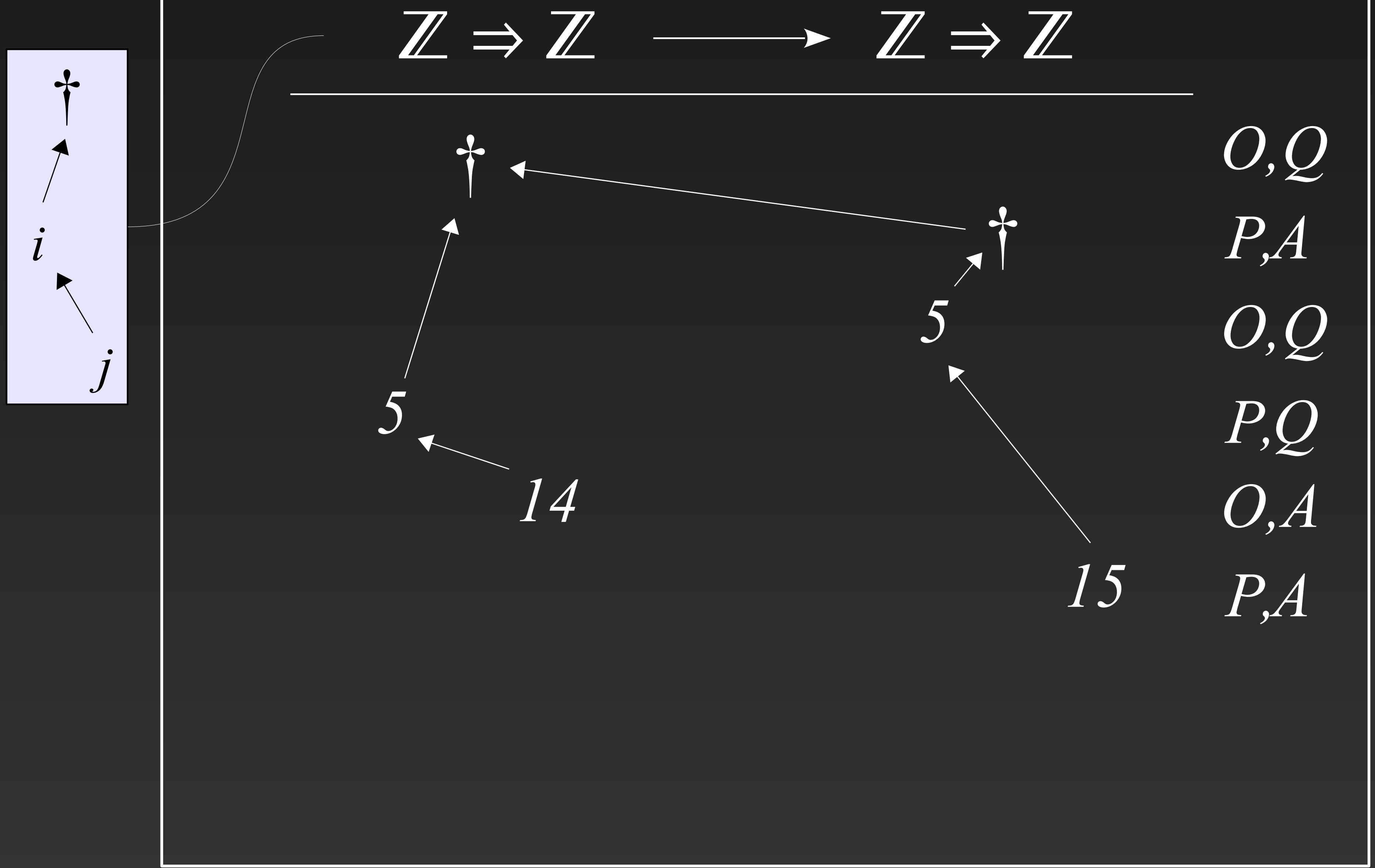
Example

$$f : \mathrm{int} \to \mathrm{int} \;\vdash\; \lambda x.\, f\,x + 1 : \mathrm{int} \to \mathrm{int}$$



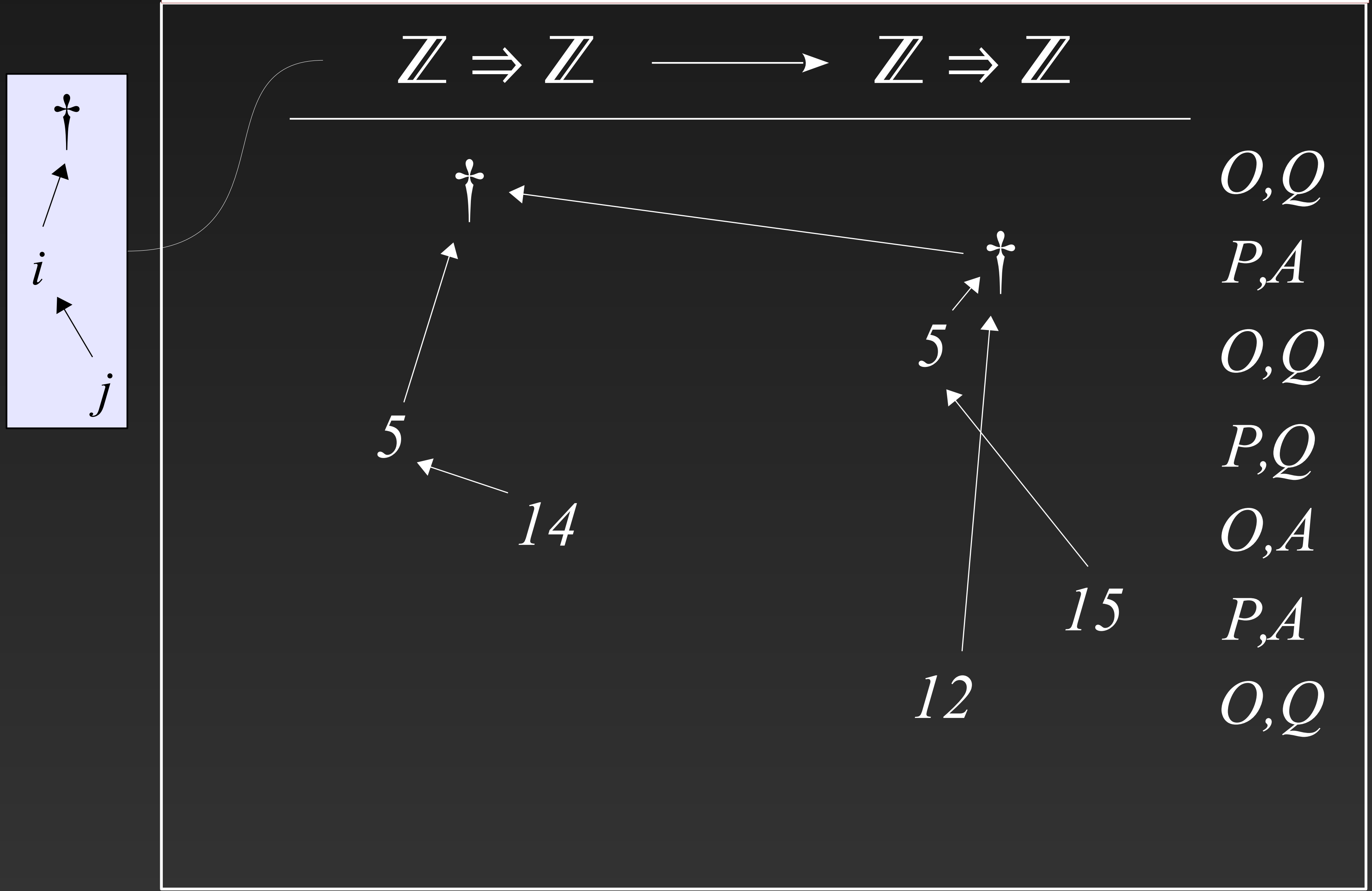
Example

$$f : \text{int} \rightarrow \text{int} \;\vdash\; \lambda x.\, f\,x + 1 : \text{int} \rightarrow \text{int}$$



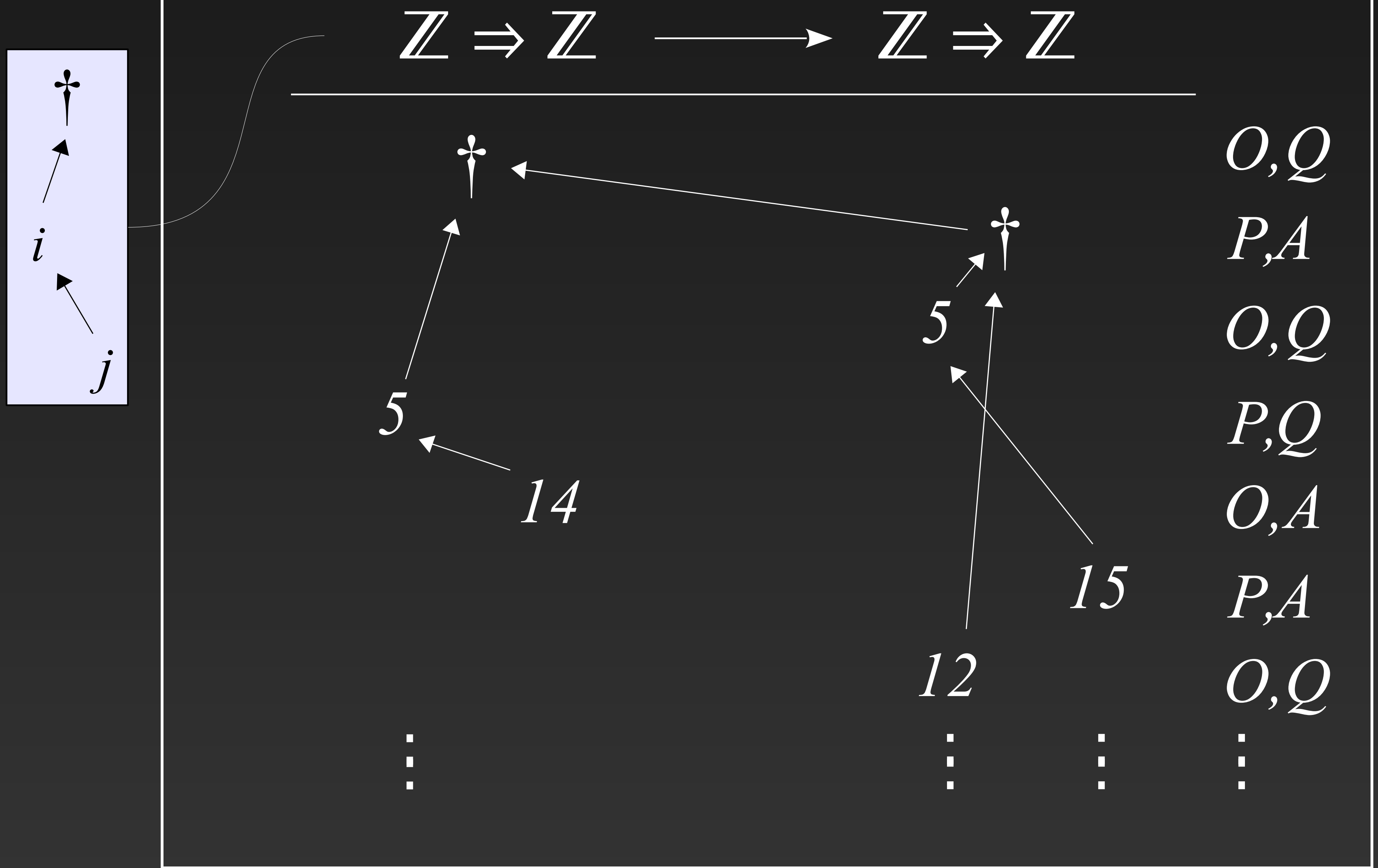
Example

$$f : \text{int} \rightarrow \text{int} \ \vdash \ \lambda x. f\,x + 1 : \text{int} \rightarrow \text{int}$$



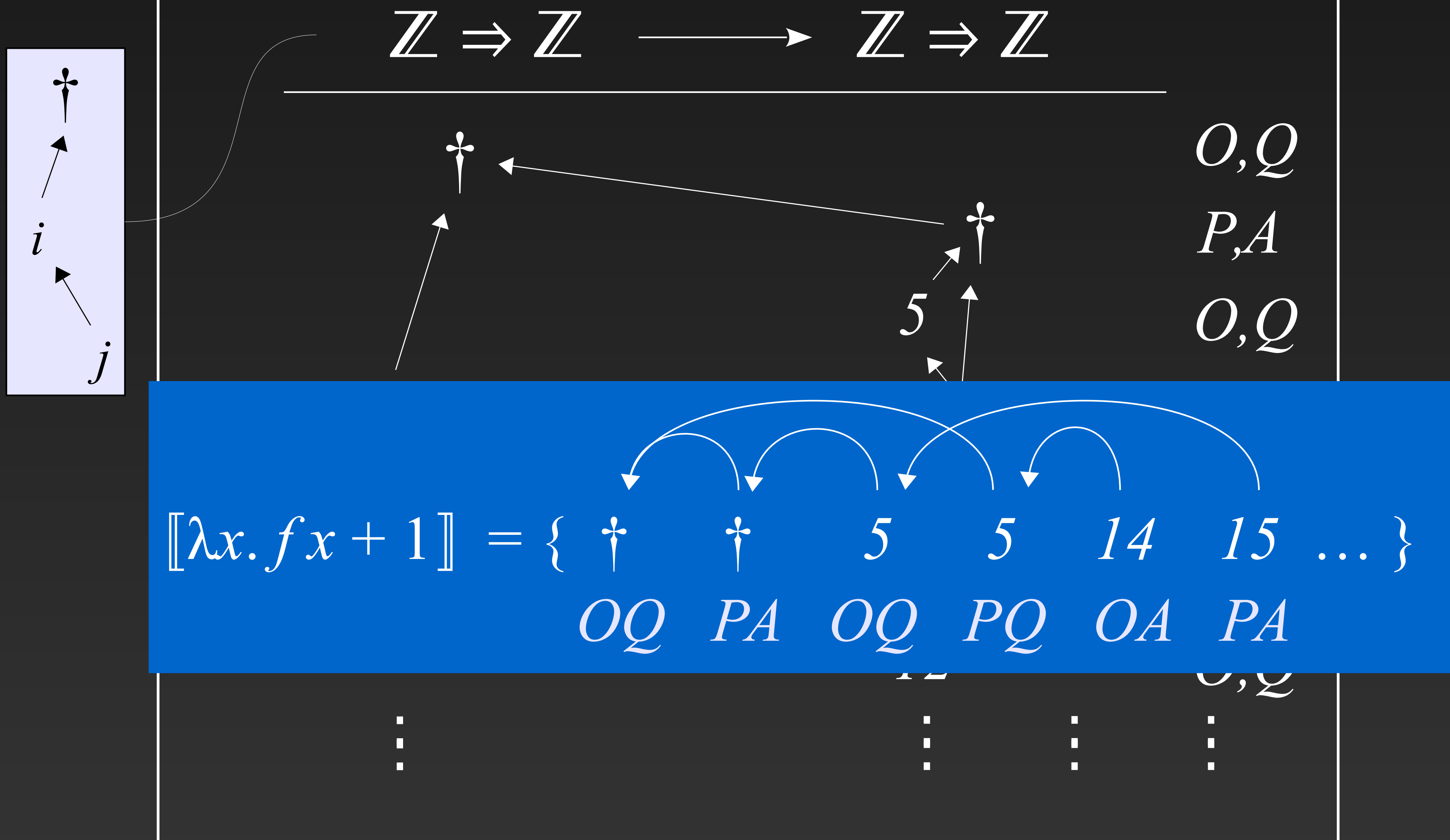
Example

$$f : \text{int} \to \text{int} \;\vdash\; \lambda x.\, f\,x + 1 : \text{int} \to \text{int}$$

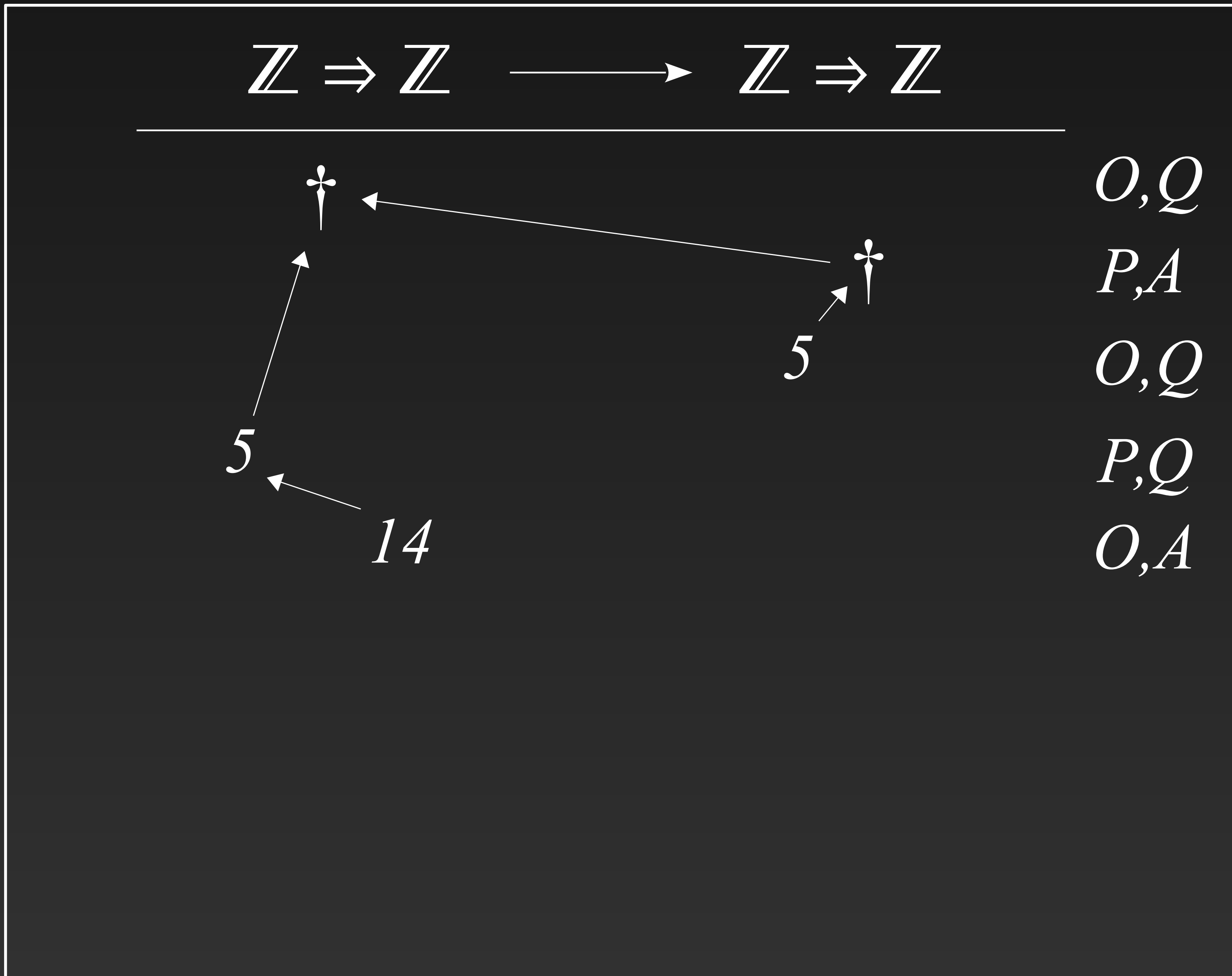


Example

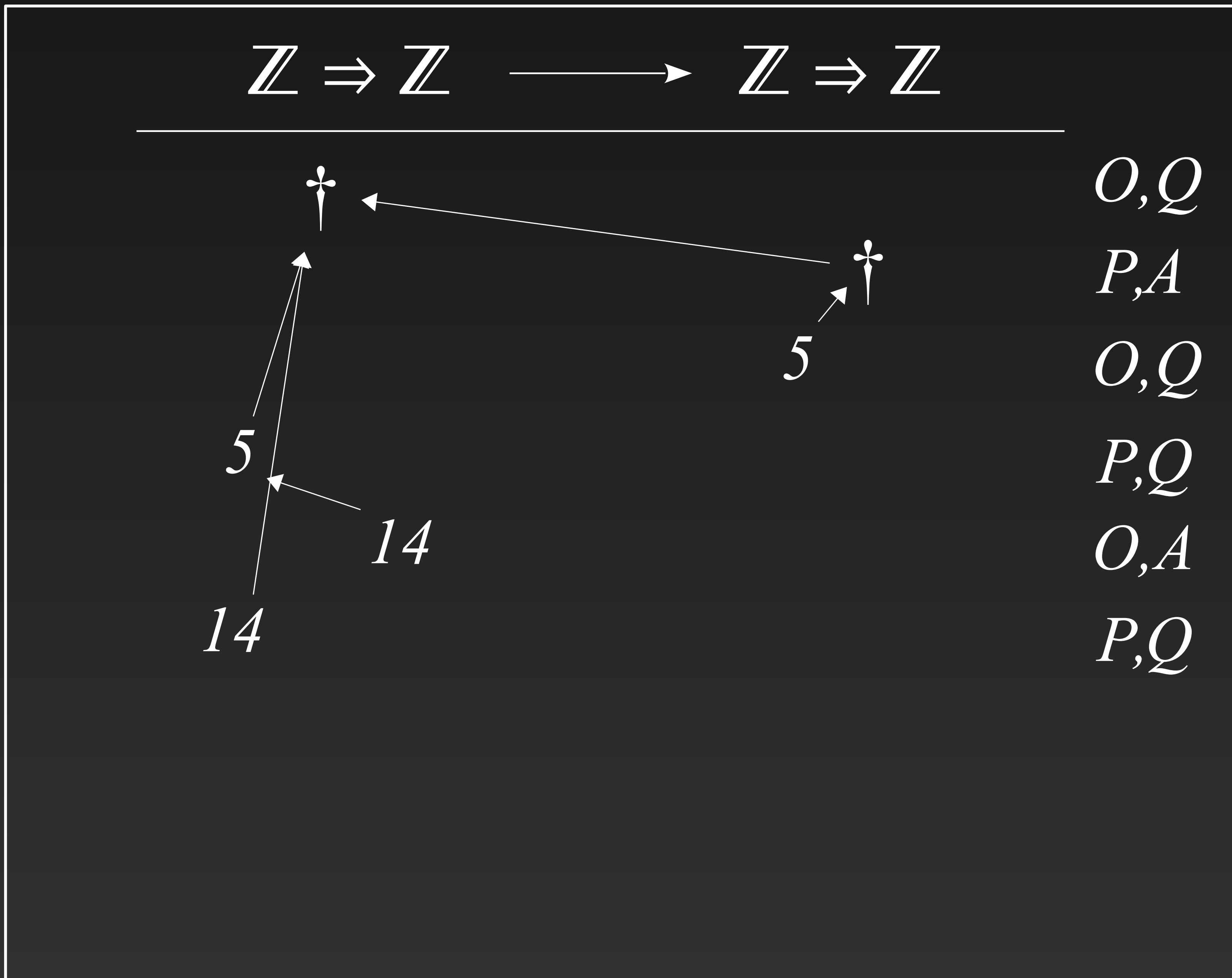
$$f : \mathrm{int} \to \mathrm{int} \ \vdash\ \lambda x. f\,x + 1 : \mathrm{int} \to \mathrm{int}$$



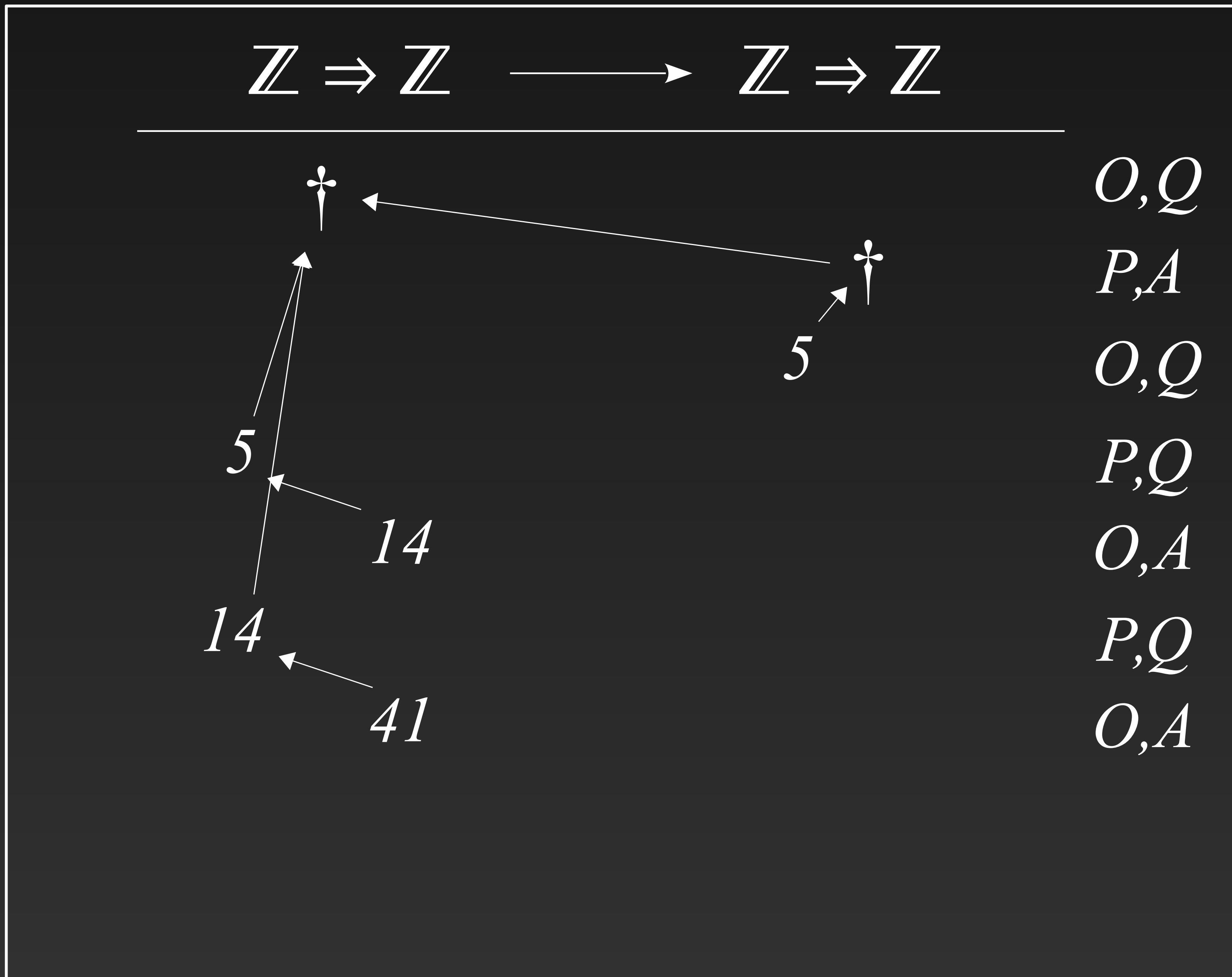
Another example



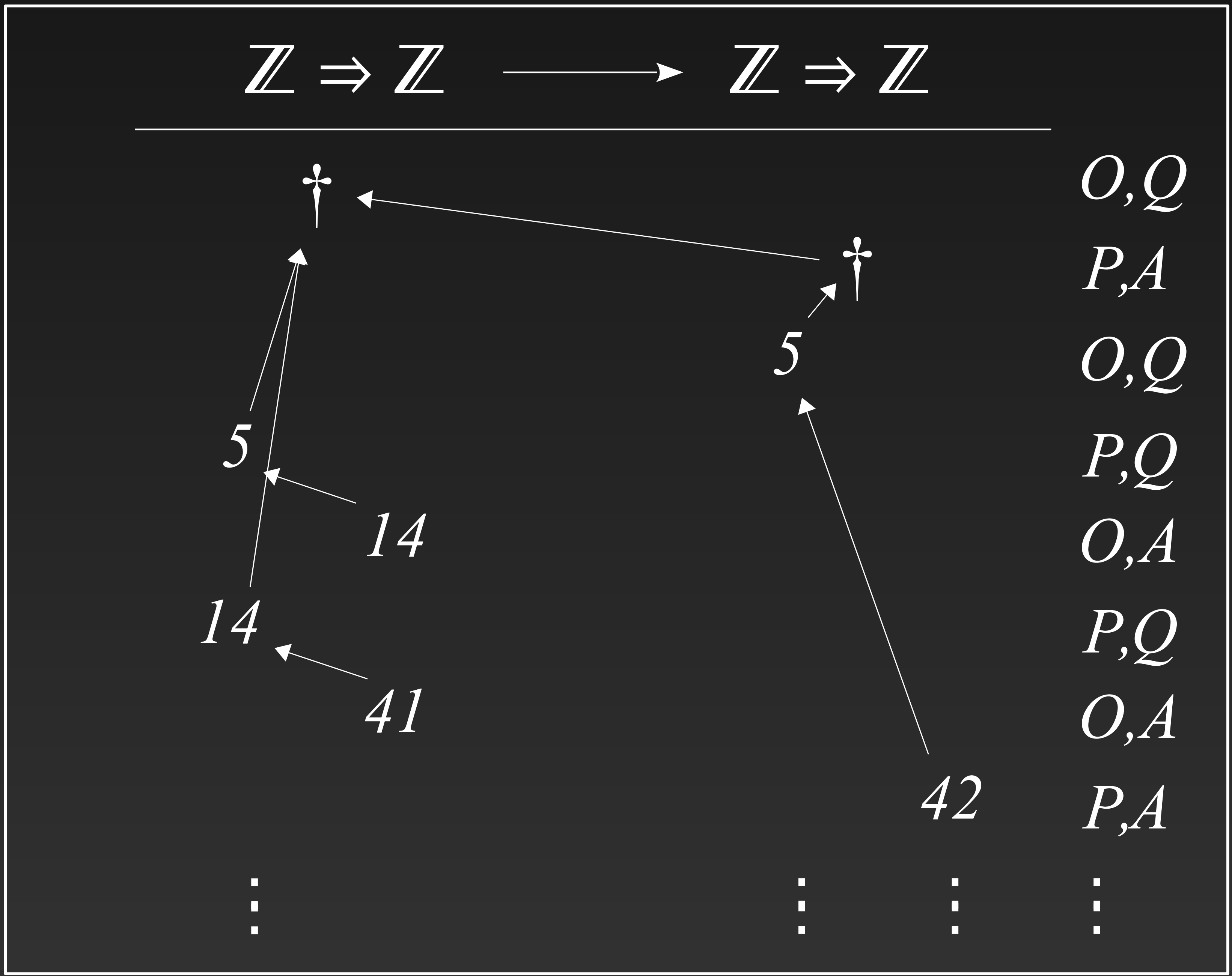
Another example



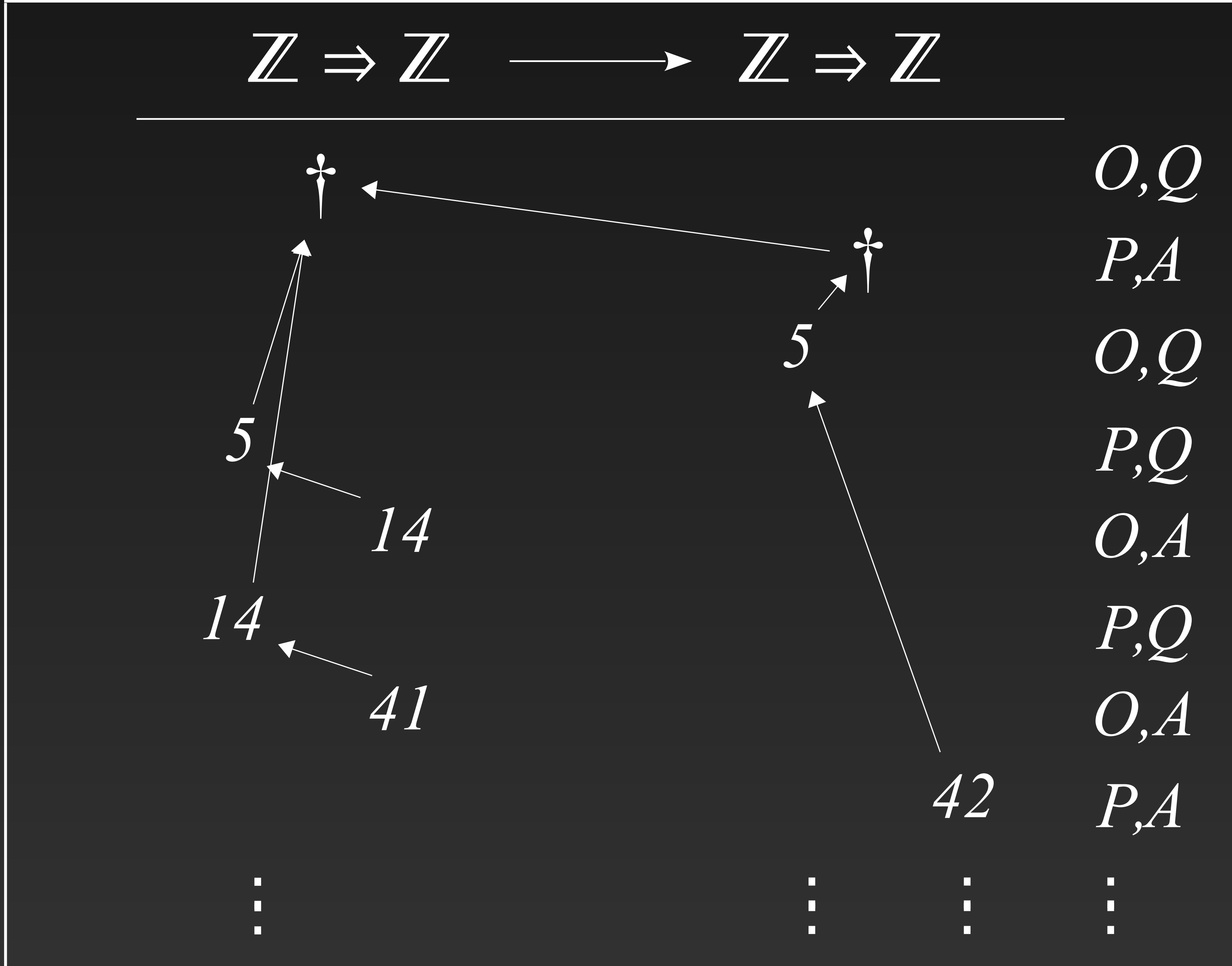
Another example



Another example



$$f : \mathrm{int} \to \mathrm{int} \;\vdash\; \lambda x.\, f(f(x)) + 1 : \mathrm{int} \to \mathrm{int}$$



Nominal games

$\vdash \mathrm{ref}() : \mathrm{ref\ unit}$

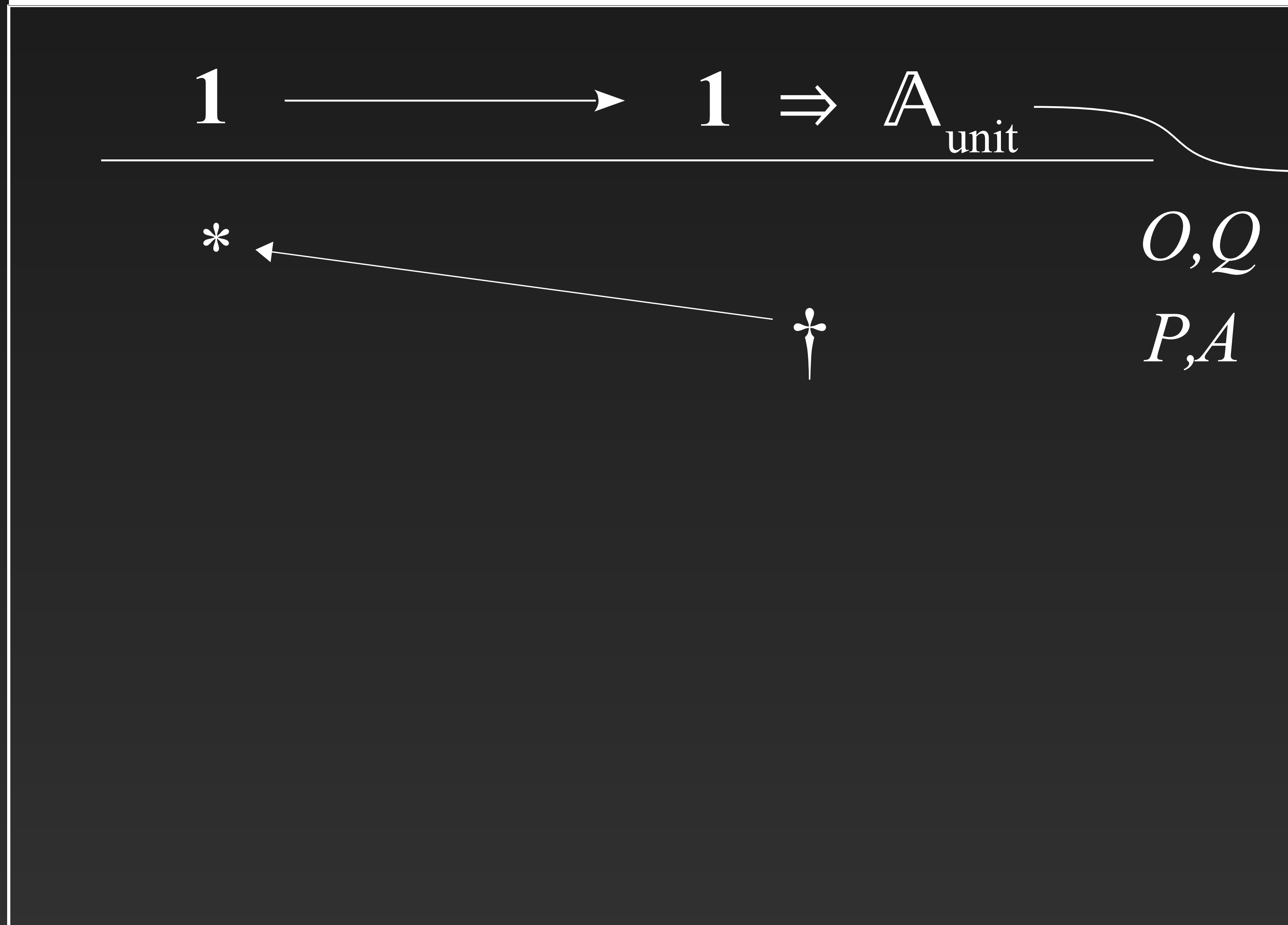
$\mathbf{1} \longrightarrow \mathbb{A}_{\mathrm{unit}}$

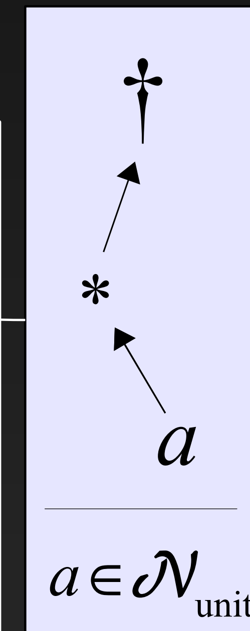
$*$		O,Q
	a	P,A

$a \in \mathcal{N}_{\mathrm{unit}}$

Nominal games (2)

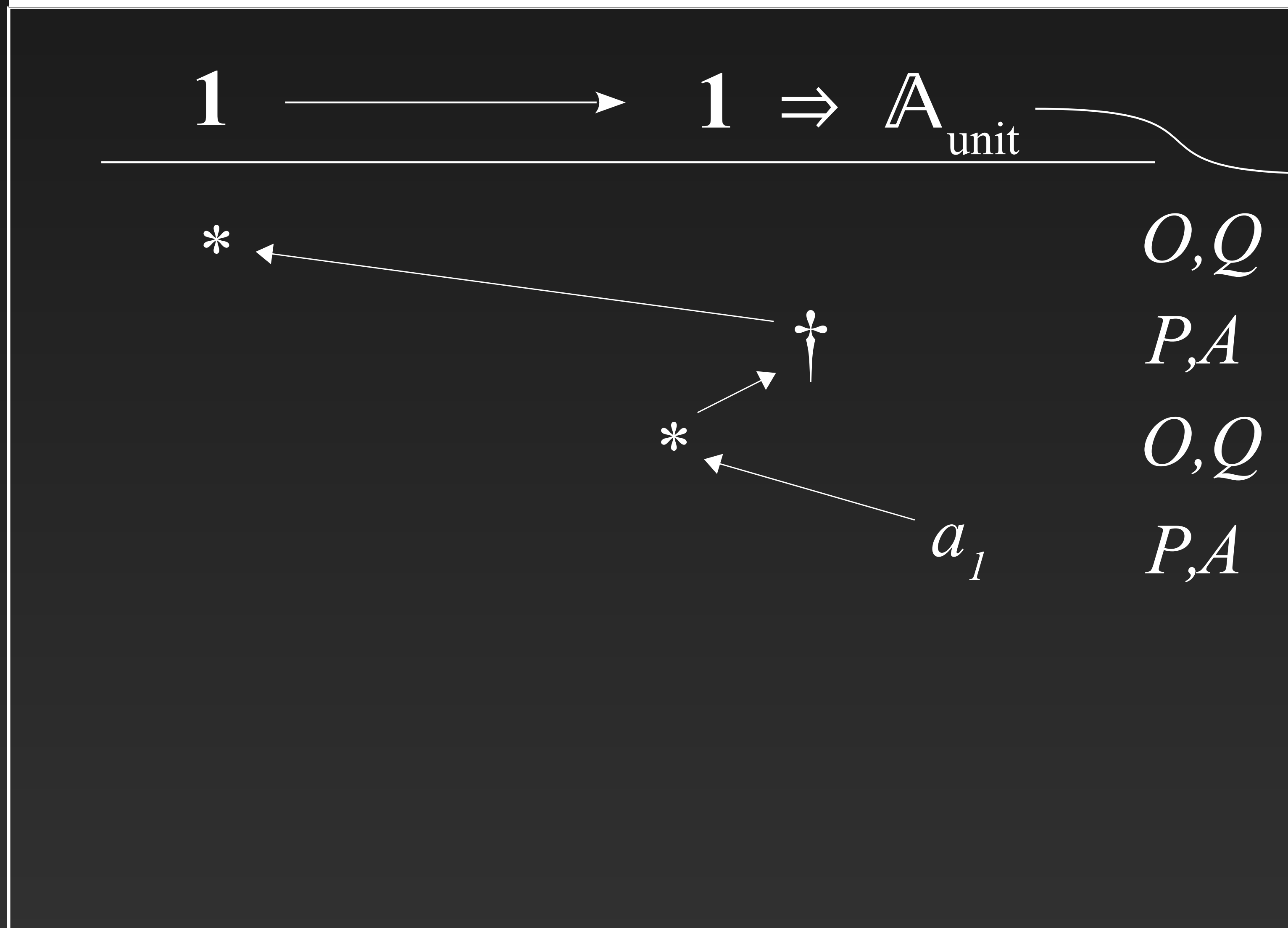
$$\vdash \lambda x.\, \mathrm{ref}() : \mathrm{unit} \rightarrow \mathrm{ref\ unit}$$

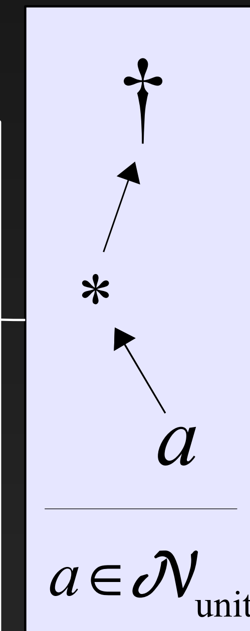




Nominal games (2)

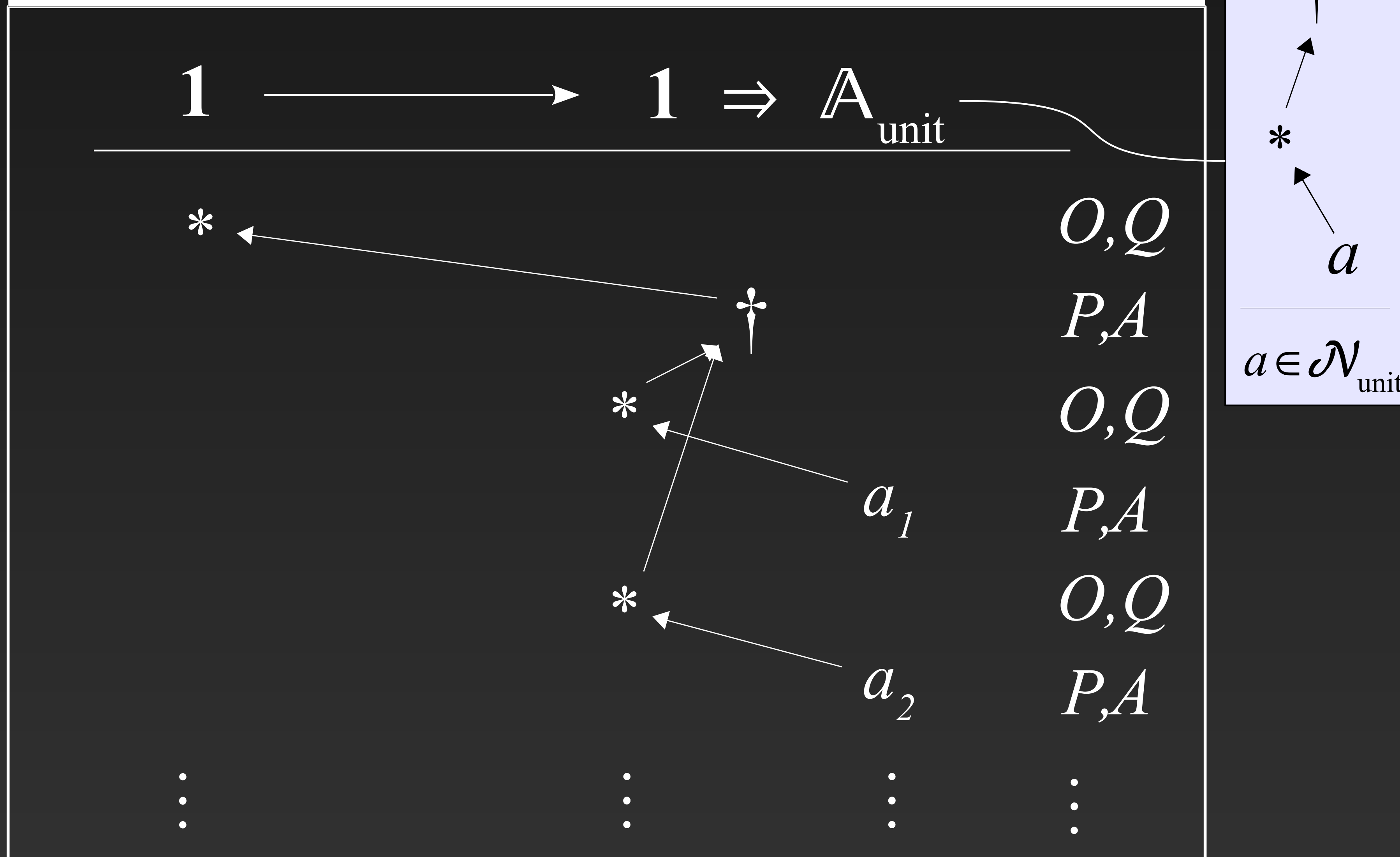
$\vdash \ \lambda x.\ \mathrm{ref}() : \mathrm{unit} \to \mathrm{ref\ unit}$





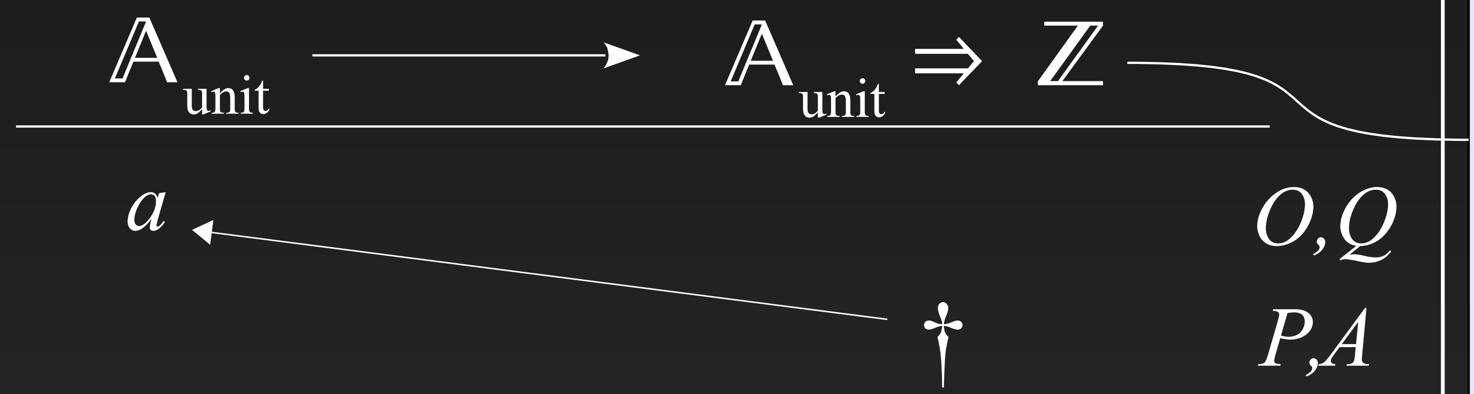
Nominal games (2)

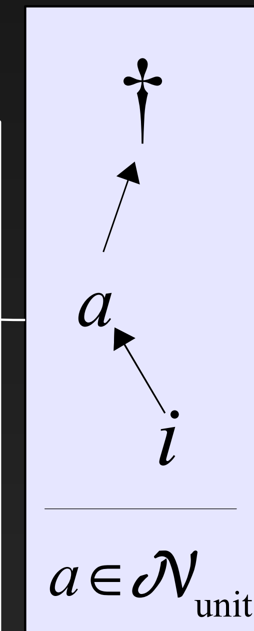
$\vdash \ \lambda x.\ \mathrm{ref}() : \mathrm{unit} \to \mathrm{ref\ unit}$



Nominal games (3)

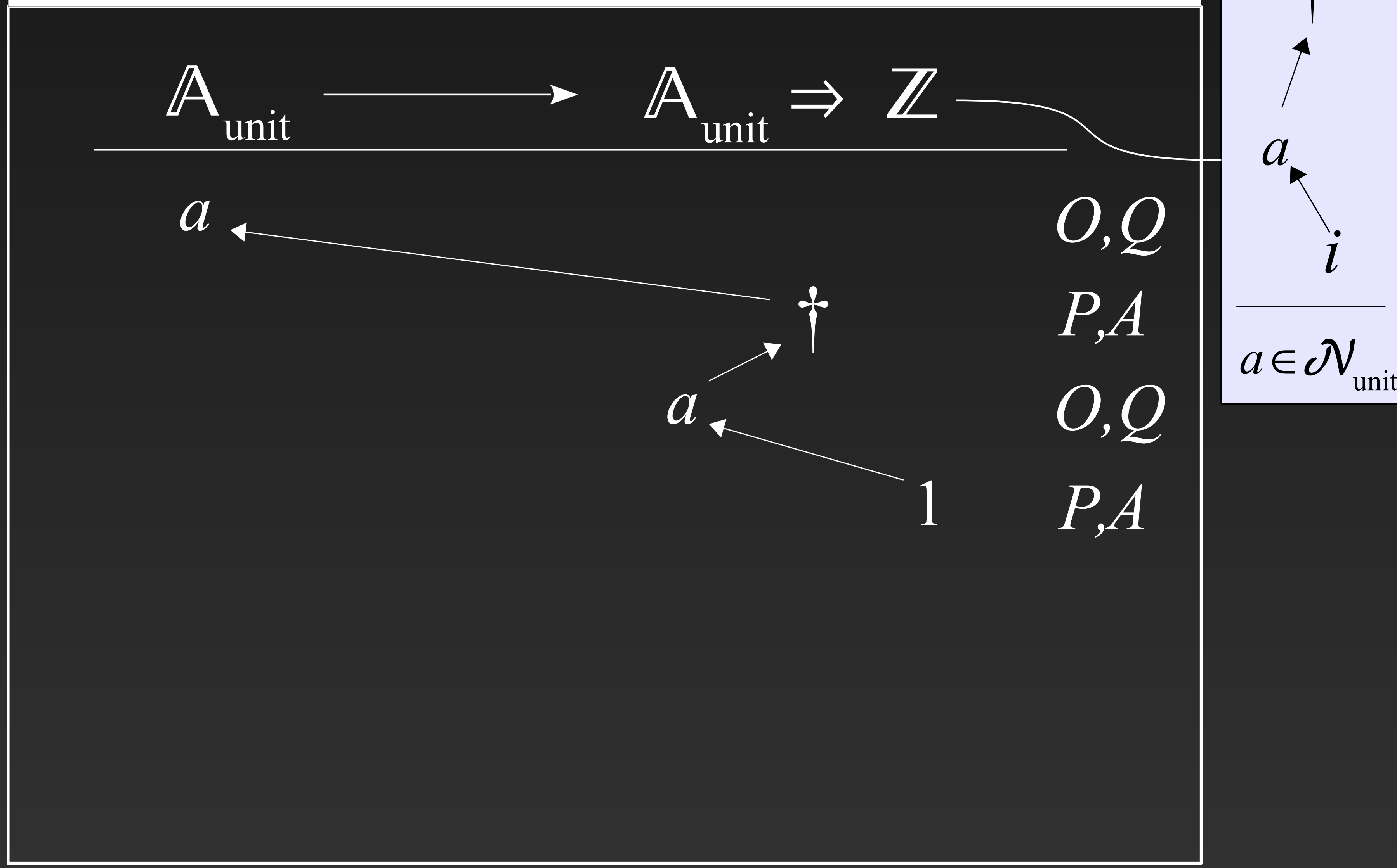
$$y : \text{ref unit} \vdash \lambda x.\, x == y : \text{ref unit} \longrightarrow \text{int}$$





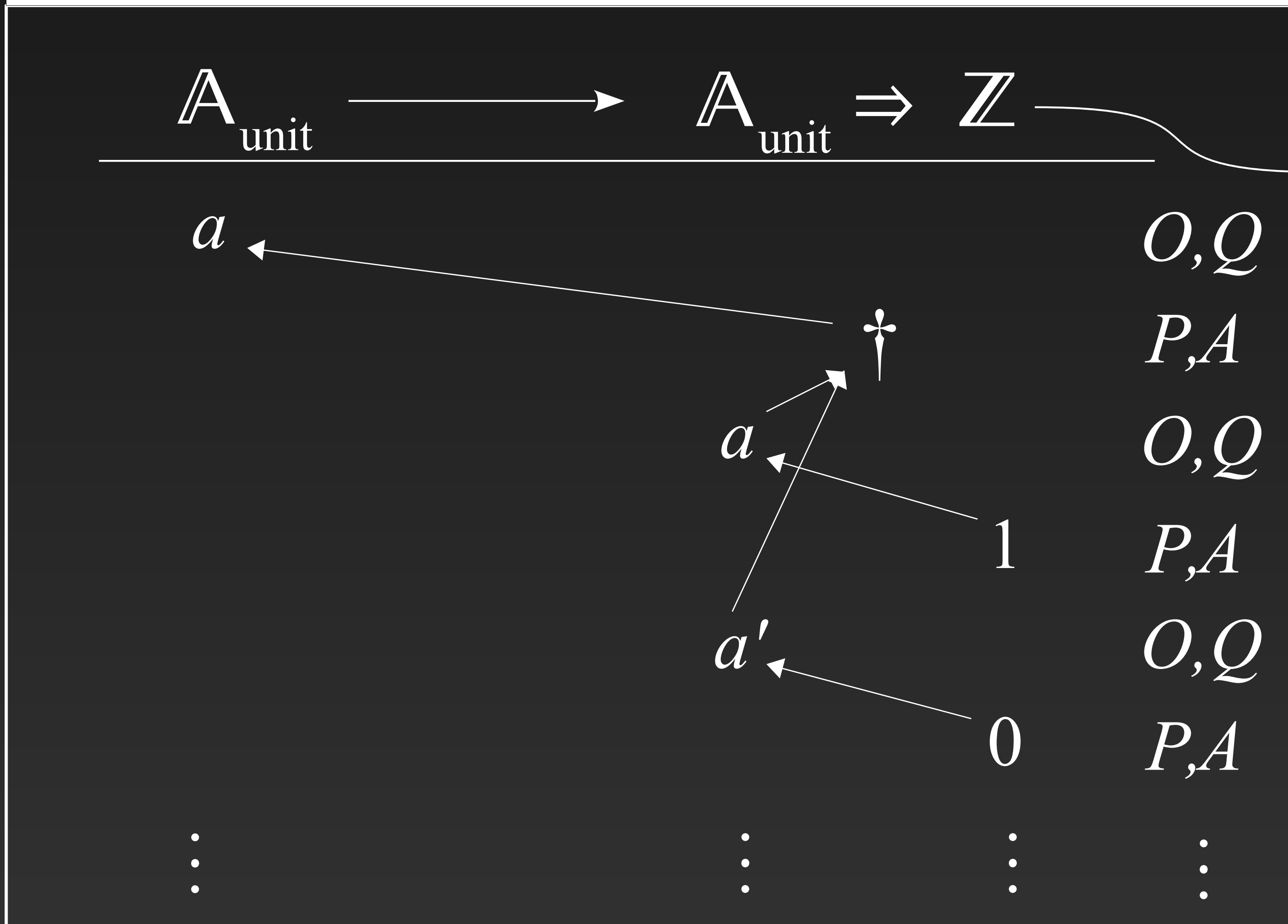
Nominal games (3)

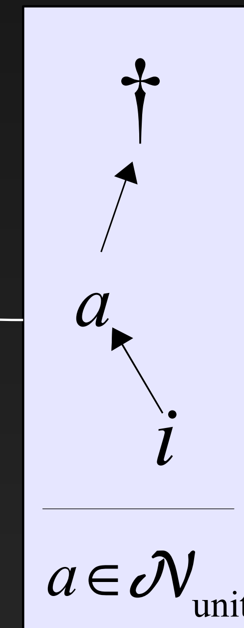
$$y : \text{ref unit} \vdash \lambda x.\ x == y : \text{ref unit} \longrightarrow \text{int}$$



Nominal games (3)

$$y : \text{ref unit} \vdash \lambda x.\ x == y : \text{ref unit} \longrightarrow \text{int}$$





Nominal games: explicit store

$$
\begin{array}{cccccccc}
a^{(a,i_0)} & \dagger^{(a,i_0)} & *^{(a,i_1)} & i_1{}^{(a,i_1)} & *^{(a,i_2)} & i_2{}^{(a,i_2)} & \cdots \\
O & P & O & P & O & P &
\end{array}
$$

$$= \llbracket x : \mathsf{int\ ref} \vdash \lambda y.\,!x\ : \mathsf{unit} \to \mathsf{int} \rrbracket : \mathbb{A}_{\mathsf{int}} \to (1 \Rightarrow \mathbb{Z})$$

Nominal games: explicit store

$$\begin{array}{ccccccc} a^{(a,i_0)} & \dagger^{(a,i_0)} & *^{(a,i_1)} & i_1{}^{(a,i_1)} & *^{(a,i_2)} & i_2{}^{(a,i_2)} & \cdots \\ O & P & O & P & O & P & \end{array}$$

$$= \; [\![x : \mathsf{int\ ref} \vdash \lambda y.\,!x \; : \mathsf{unit} \to \mathsf{int}]\!] : \mathbb{A}_{\mathsf{int}} \to (1 \Rightarrow \mathbb{Z})$$

$$\begin{array}{cccccccc} a^{(a,\dagger)} & \dagger^{(a,\dagger)} & *^{(a,\dagger)} & *^{(a,\dagger)} & *^{(a,\dagger)} & 3^{(a,\dagger)} & i^{(a,\dagger)} & i^{(a,\dagger)} \\ O & P & O & P & O & P & O & P \end{array}$$

$$= \; [\![x : (\mathsf{unit} \to \mathsf{int})\ \mathsf{ref} \vdash x := \lambda y.3;\ \lambda y.(!x)()]\!] : \mathbb{A}_{\mathsf{unit}\to\mathsf{int}} \to (1 \Rightarrow \mathbb{Z})$$

In LaTeX

Definition An ***arena*** $A = (M_A, I_A, \vdash_A, \lambda_A)$ is given by:

- a strong nominal set M_A of *moves*,
- a nominal subset $I_A \subseteq M_A$ of *initial moves*,
- a nominal *labelling* function $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$,
- a nominal *justification* relation $\vdash_A \subseteq M_A \times (M_A \setminus I_A)$;

satisfying the conditions:

- $m \in I_A \implies \lambda_A(m) = (P, A)$
- $m \vdash_A n \implies \lambda_A^{O,P}(m) \neq \lambda_A^{O,P}(n) \wedge (\lambda_A^{Q,A}(m) = A \implies \lambda_A^{Q,A}(n) = Q)$

Games live in a category of *nominal sets* [Gabbay&Pitts]

In LaTeX

Games live in a category of *nominal sets* [Gabbay&Pitts]

Definition An ***arena*** $A = (M_A, I_A, \vdash_A, \lambda_A)$ is given by:

- a strong nominal set M_A of *moves*,
- a nominal subset $I_A \subseteq M_A$ of *initial moves*,
- a nominal *labelling* function $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$,
- a nominal *justification* relation $\vdash_A \subseteq M_A \times (M_A \setminus I_A)$;

satisfying the conditions:

- $m \in I_A \implies \lambda_A(m) = (P, A)$
- $m \vdash_A n \implies \lambda_A^{O,P}(m) \neq \lambda_A^{O,P}(n) \wedge (\lambda_A^{Q,A}(m) = A \implies \lambda_A^{Q,A}(n) = Q)$

Given arenas A, B, the ***interface*** $A \to B = (M_{A\to B}, I_{A\to B}, \vdash_{A\to B}, \lambda_{A\to B})$ is given by:

- $M_{A\to B} = M_A \uplus M_B$ and $I_{A\to B} = I_A$
- $\lambda_{A\to B} = [\lambda_A, \lambda_B]$
- $\vdash_{A\to B} = (I_A \times I_B) \cup \vdash_A \cup \vdash_B$

In LaTeX

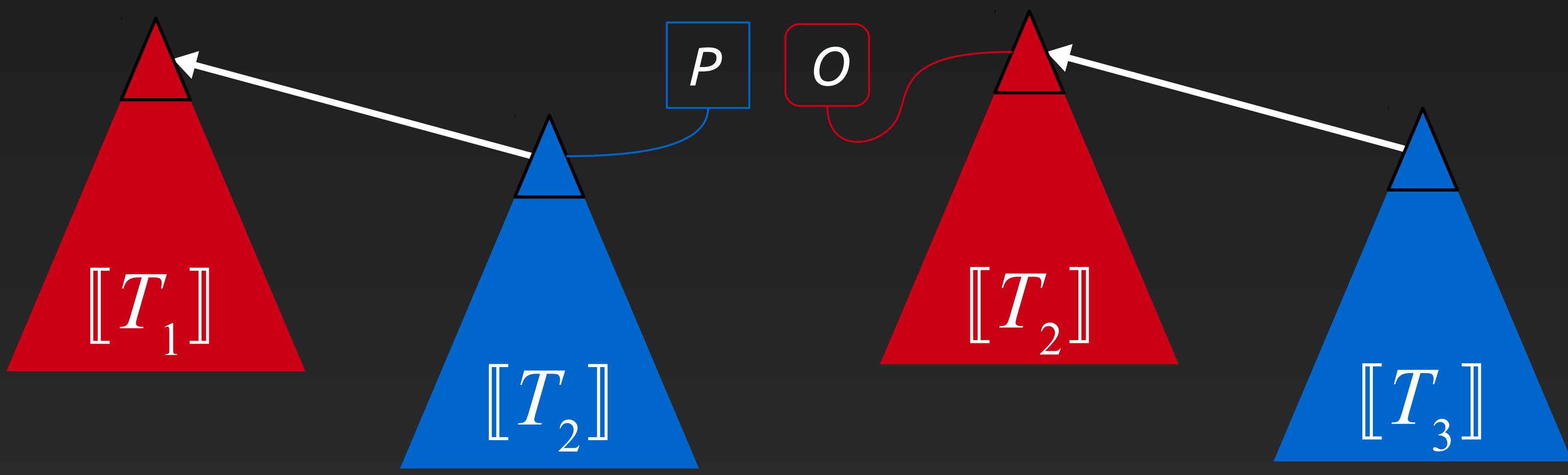
Definition A ***play*** in $A \to B$ is a sequence of *moves-with-store*, written m^{Σ}, where $m \in M_{A \to B}$ and Σ a finite *store*, satisfying the conditions:

- O/P alternation,
- justification,
- well-bracketing,
- frugality (stores contain only visible/reachable names),
- ...

Definition Given an interface $A \to B$, a ***strategy*** $\sigma : A \to B$ is a nominal set of even-length plays satisfying the conditions: *[. . .]*

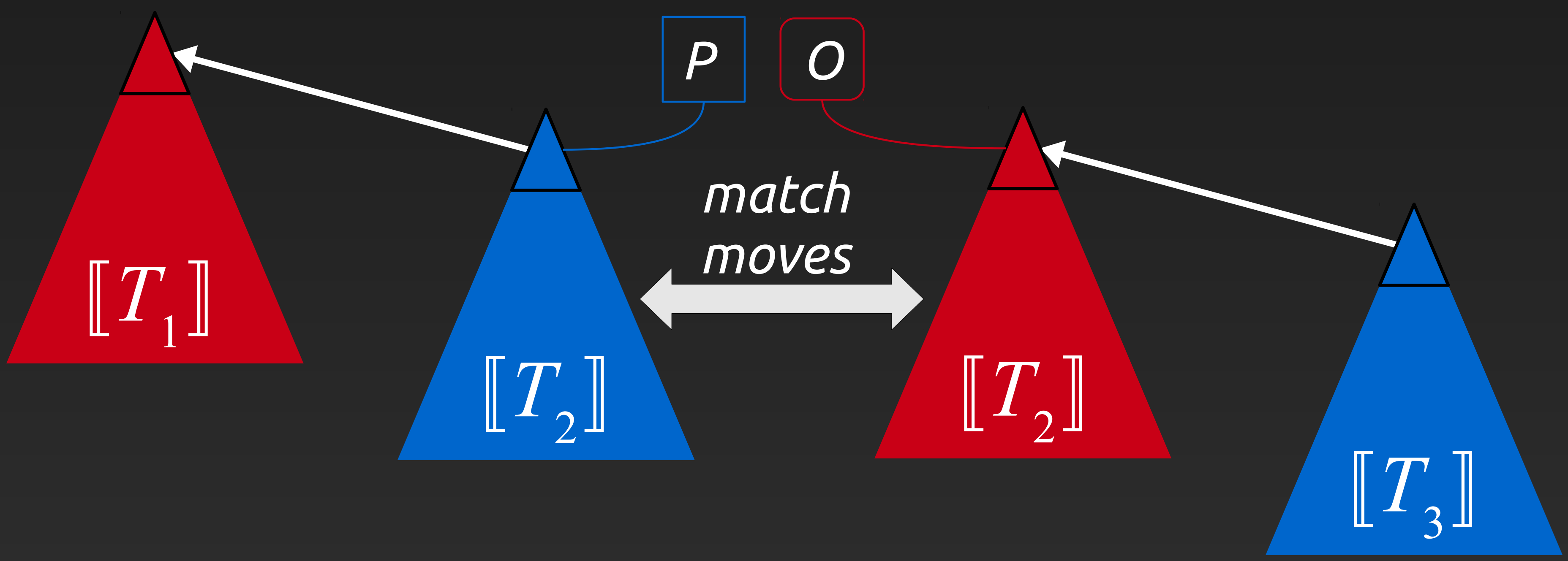
Composition

$$[\![f_1 : T_1 \vdash M_1 : T_2]\!] \qquad [\![f_2 : T_2 \vdash M_2 : T_3]\!]$$



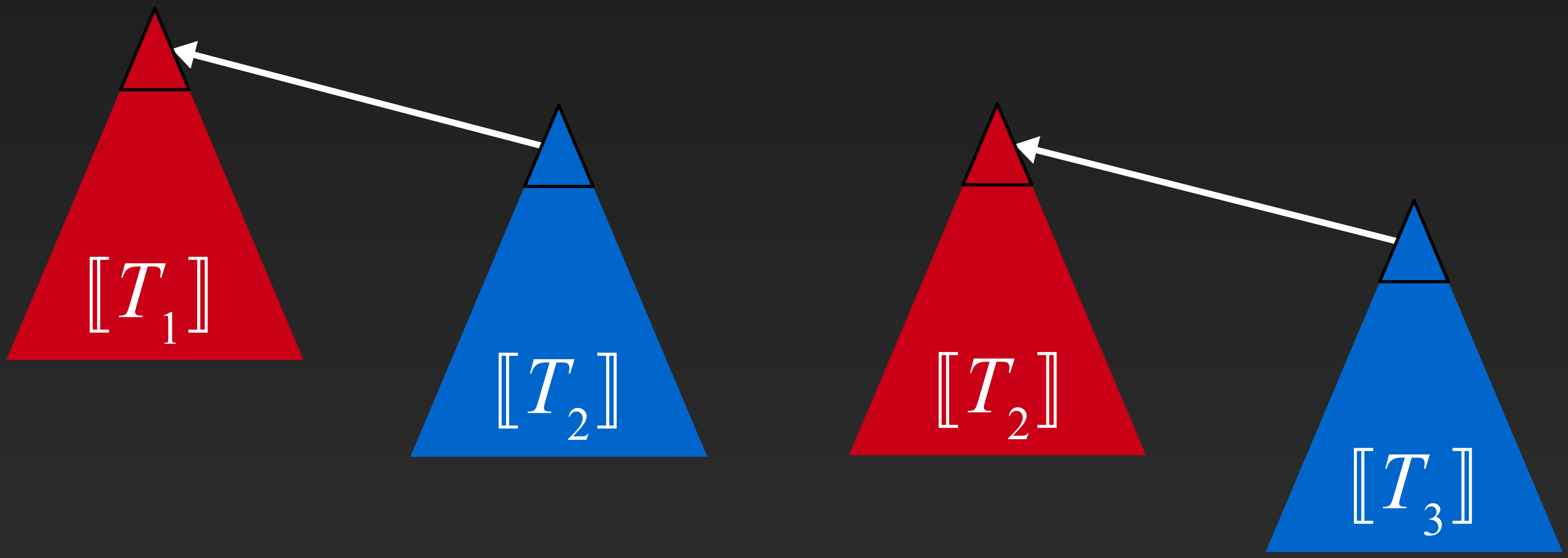
Composition

$$[\![f_1 : T_1 \vdash M_1 : T_2]\!] \qquad [\![f_2 : T_2 \vdash M_2 : T_3]\!]$$



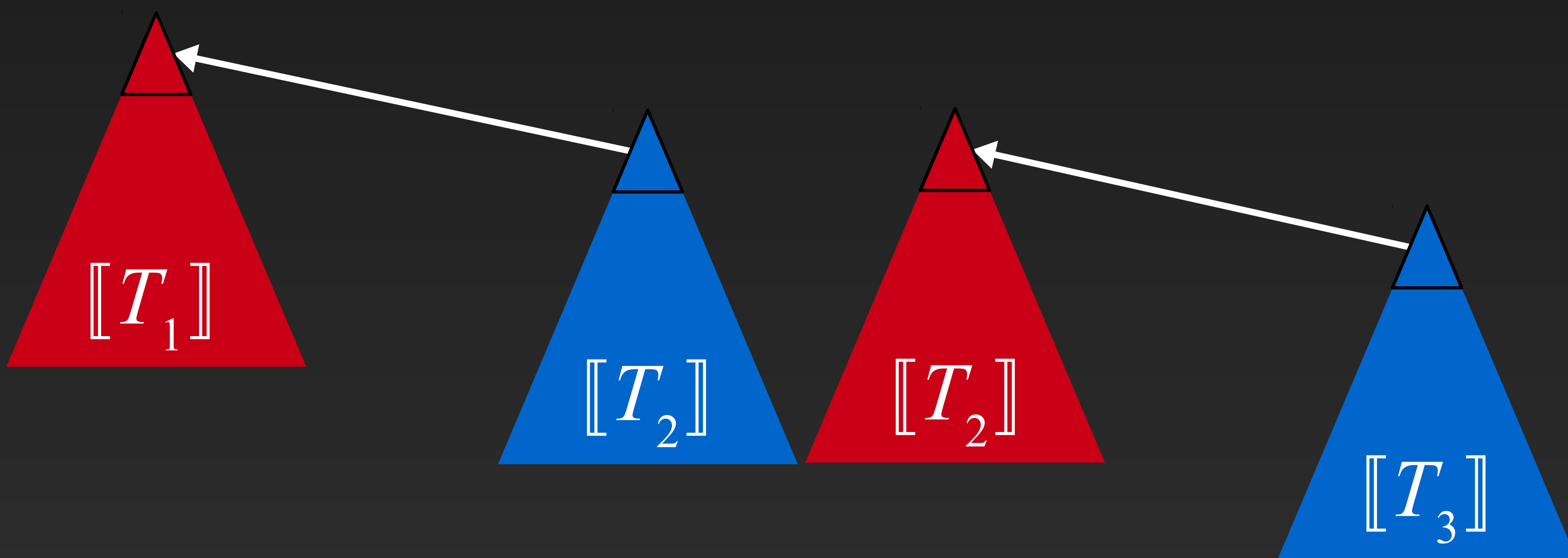
Composition

$$[\![f_1 : T_1 \vdash M_1 : T_2]\!] \qquad [\![f_2 : T_2 \vdash M_2 : T_3]\!]$$



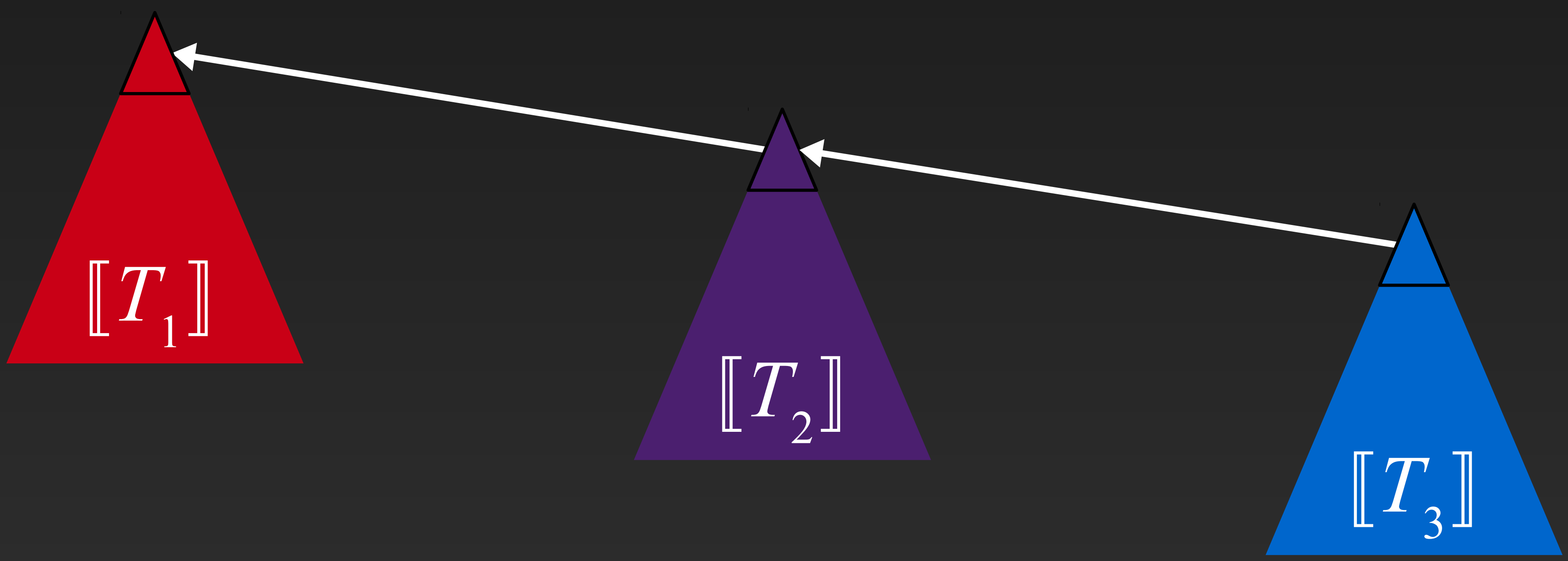
Composition

$$[\![f_1 : T_1 \vdash M_1 : T_2]\!] \qquad [\![f_2 : T_2 \vdash M_2 : T_3]\!]$$



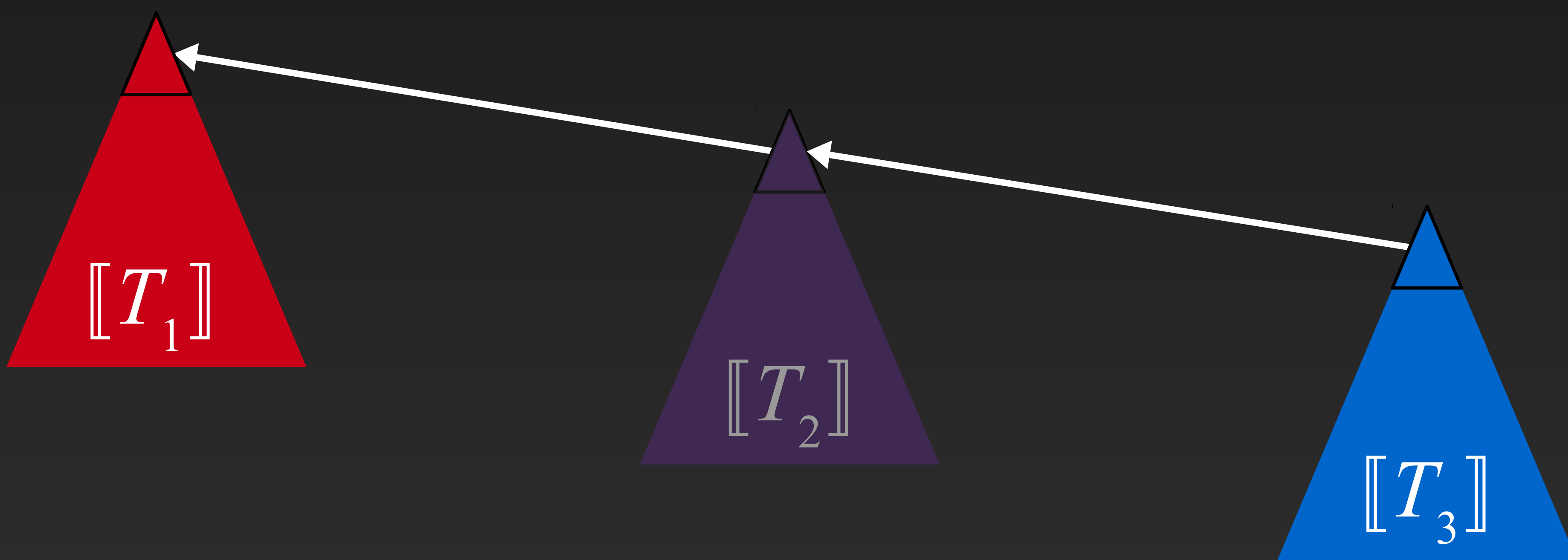
Composition

$$[\![f_1 : T_1 \vdash M_1 : T_2]\!] \qquad [\![f_2 : T_2 \vdash M_2 : T_3]\!]$$



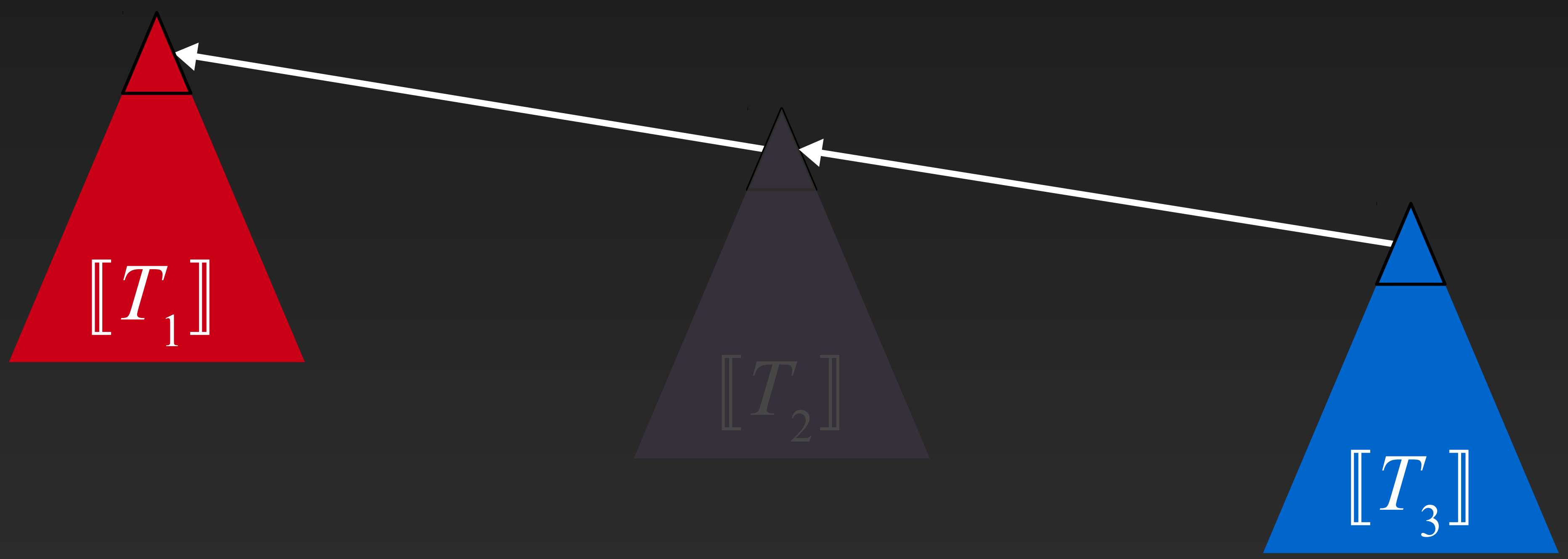
Composition

$[\![f_1 : T_1 \vdash M_1 : T_2]\!]$ $[\![f_2 : T_2 \vdash M_2 : T_3]\!]$



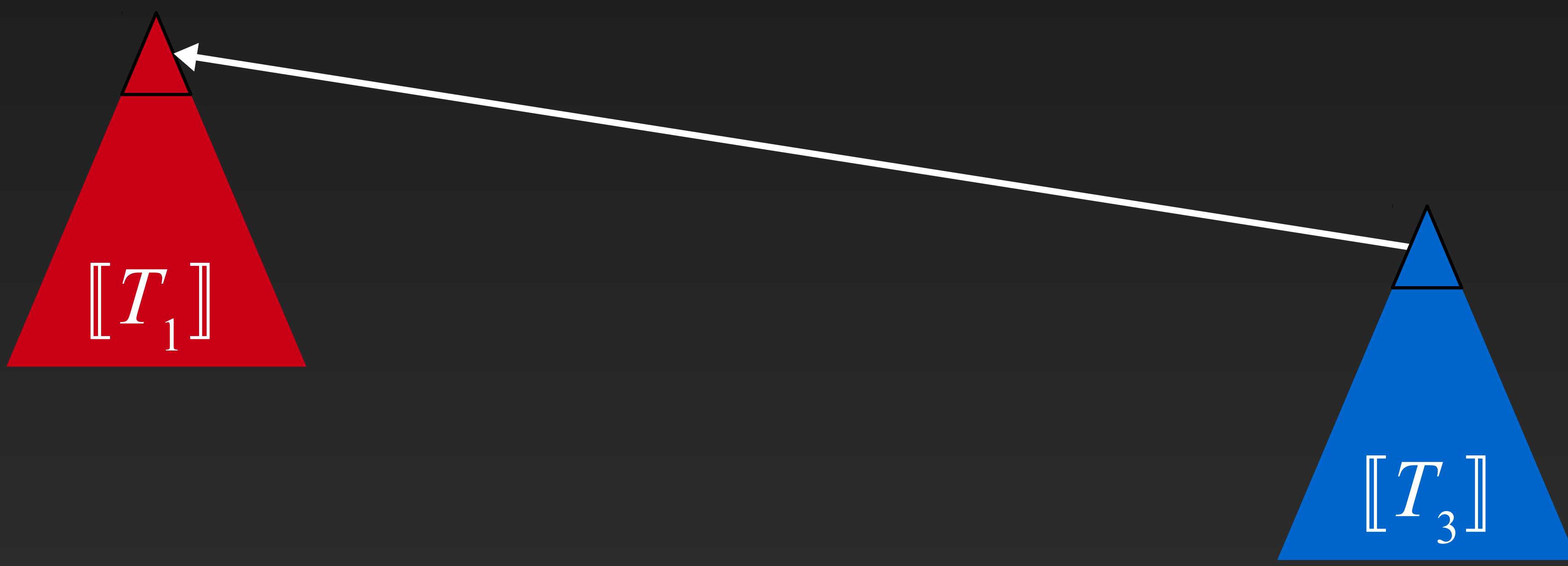
Composition

$$[\![f_1 : T_1 \vdash M_1 : T_2]\!] \qquad [\![f_2 : T_2 \vdash M_2 : T_3]\!]$$



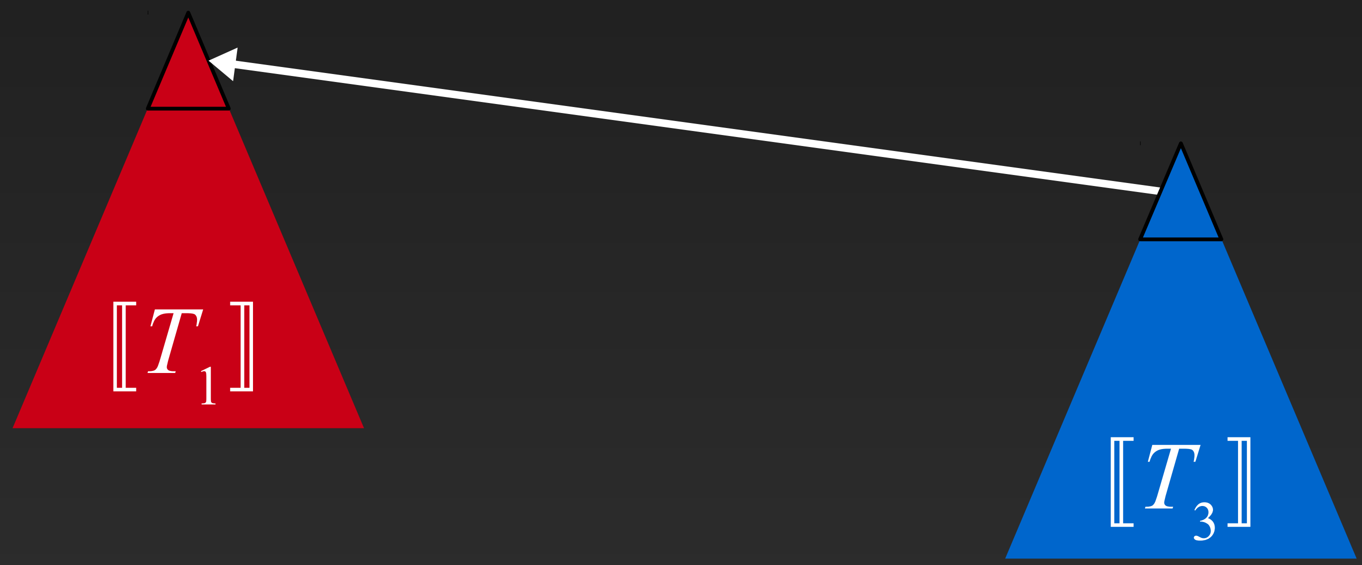
Composition

$$[\![f_1{:}T_1 \vdash M_1{:}T_2]\!] \qquad [\![f_2{:}T_2 \vdash M_2{:}T_3]\!]$$



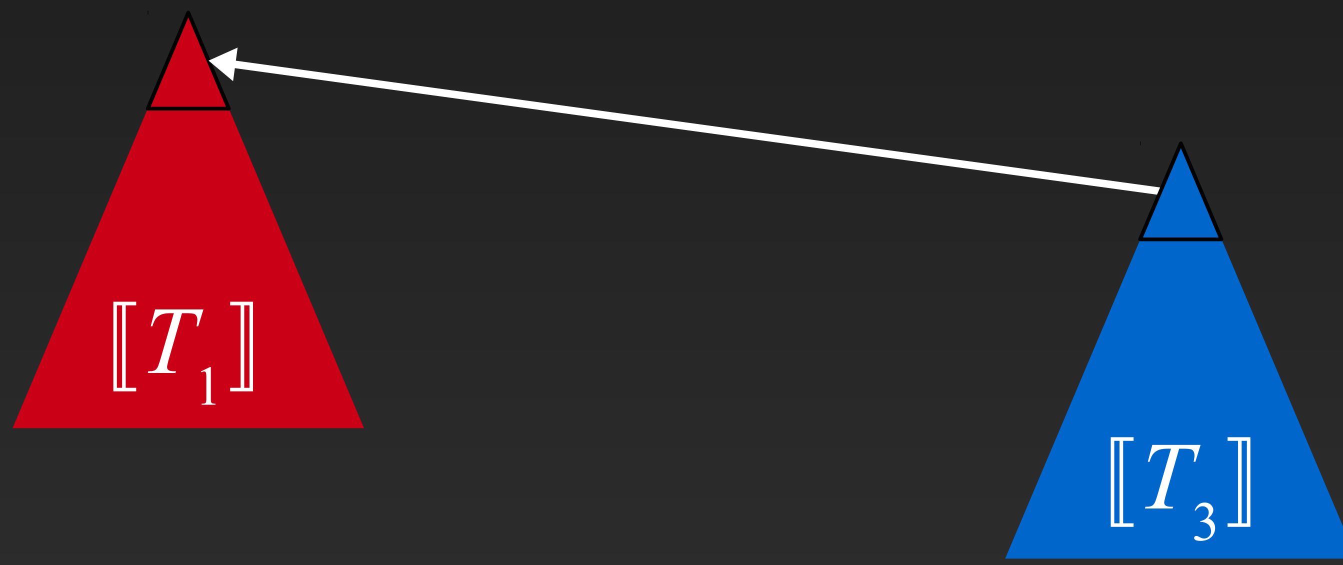
Composition

$$[\![f_1{:}T_1 \vdash M_1{:}T_2]\!] \;\; ; \;\; [\![f_2{:}T_2 \vdash M_2{:}T_3]\!]$$
$$= [\![f_1{:}T_1 \vdash \text{let}\, f_2 = M_1 \text{ in } M_2{:}T_3]\!]$$



Composition

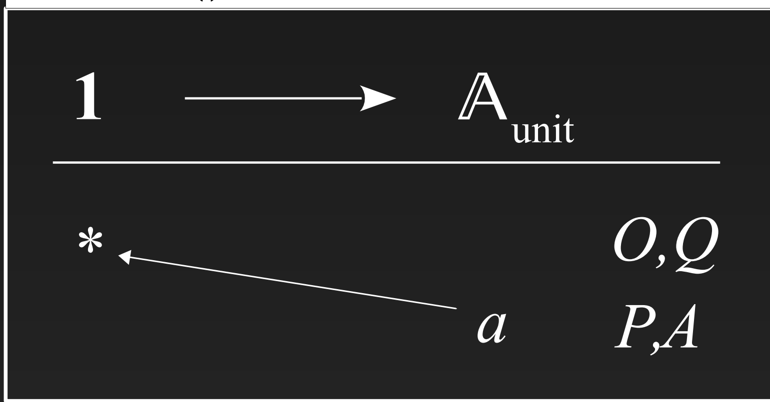
$$[\![f_1{:}T_1 \vdash M_1{:}T_2]\!] \;\; ; \;\; [\![f_2{:}T_2 \vdash M_2{:}T_3]\!]$$
$$= [\![f_1{:}T_1 \vdash \text{let}\, f_2 = M_1 \text{ in } M_2{:}T_3]\!]$$



$$A \xrightarrow{\sigma} B \xrightarrow{\tau} C \quad = \quad A \xrightarrow{\sigma;\tau} C$$

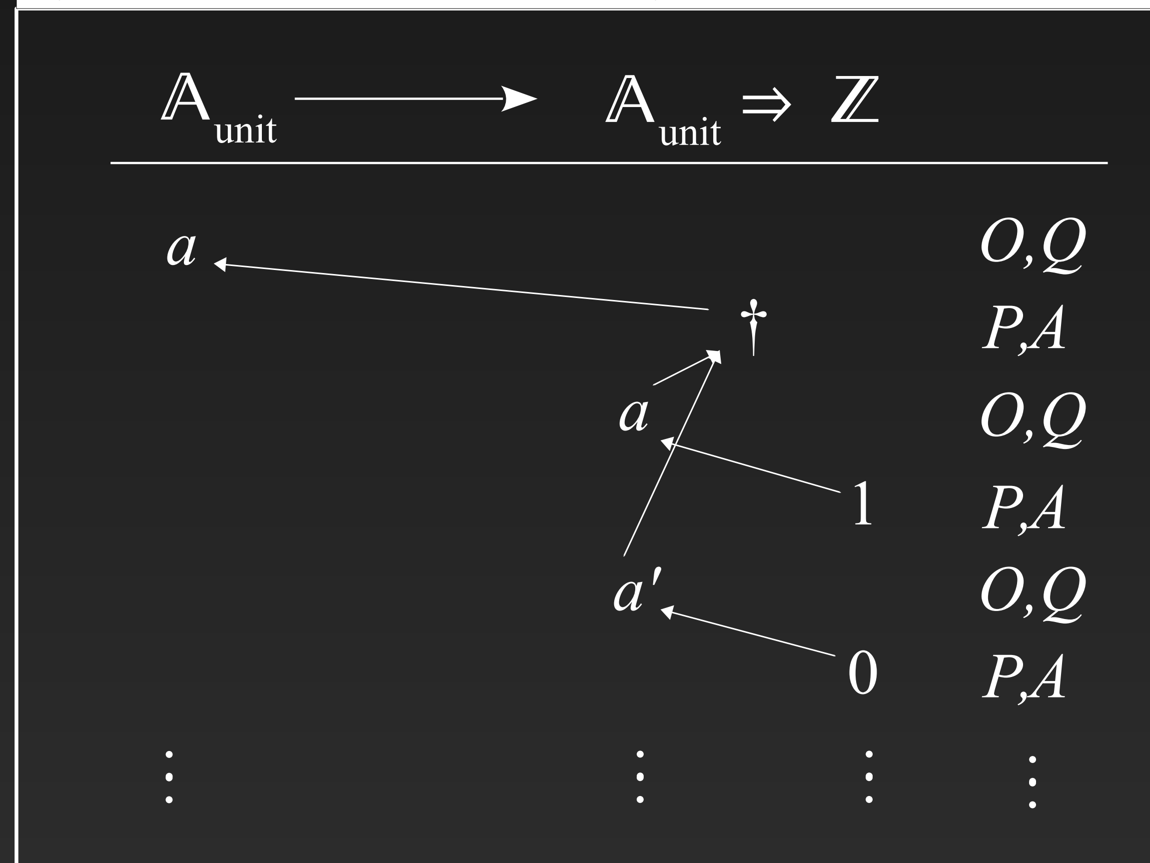
Nominal composition example

$\vdash \mathrm{ref}() : \mathrm{ref\ unit}$



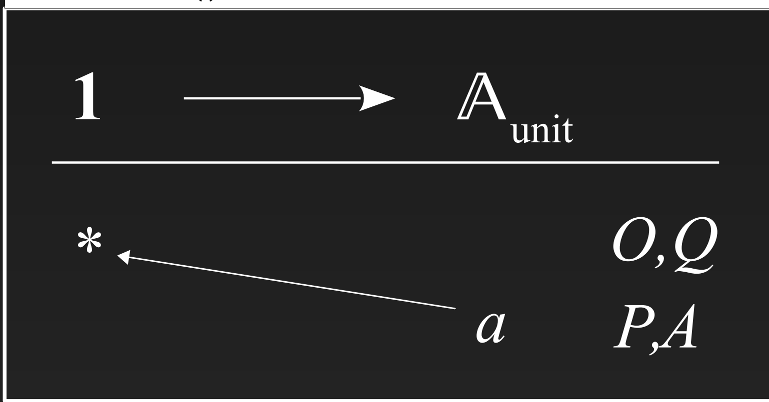
;

$y : \mathrm{ref\ unit} \vdash \lambda x.\, x == y : \mathrm{ref\ unit} \to \mathrm{int}$



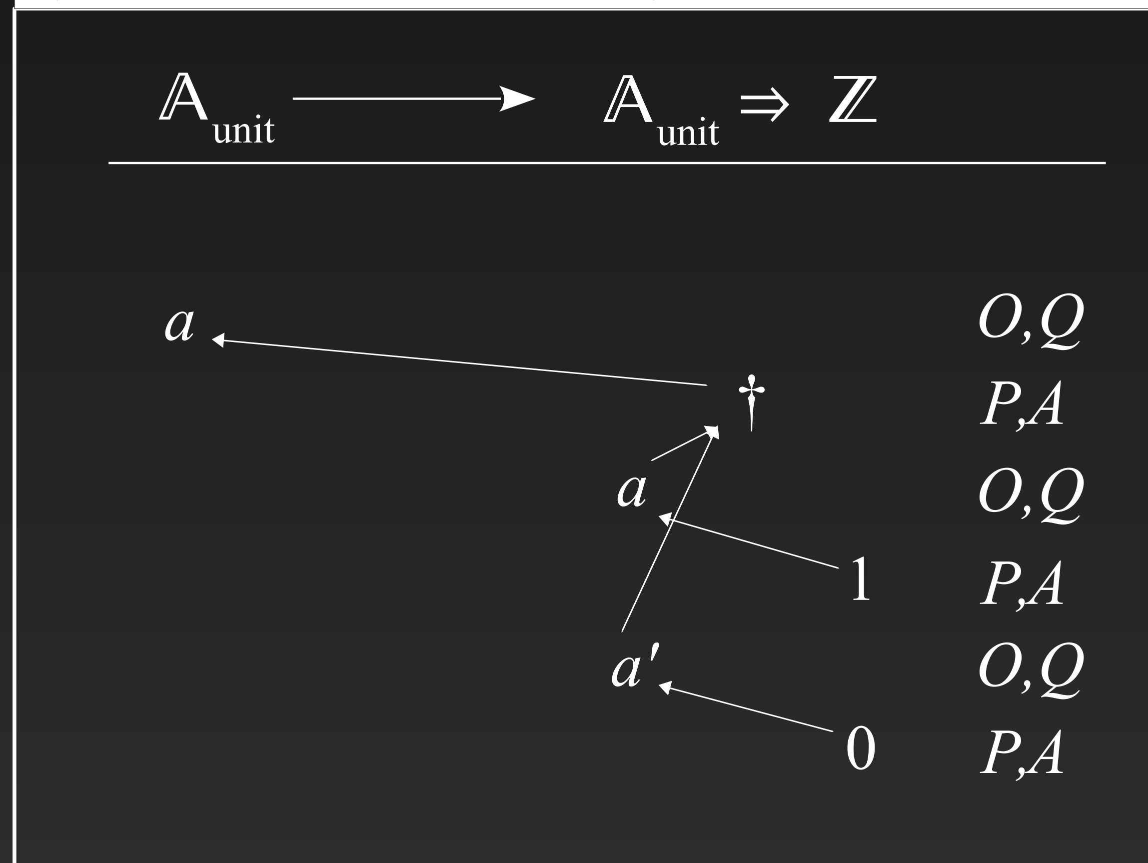
Nominal composition example

$\vdash \mathrm{ref}() : \mathrm{ref\ unit}$

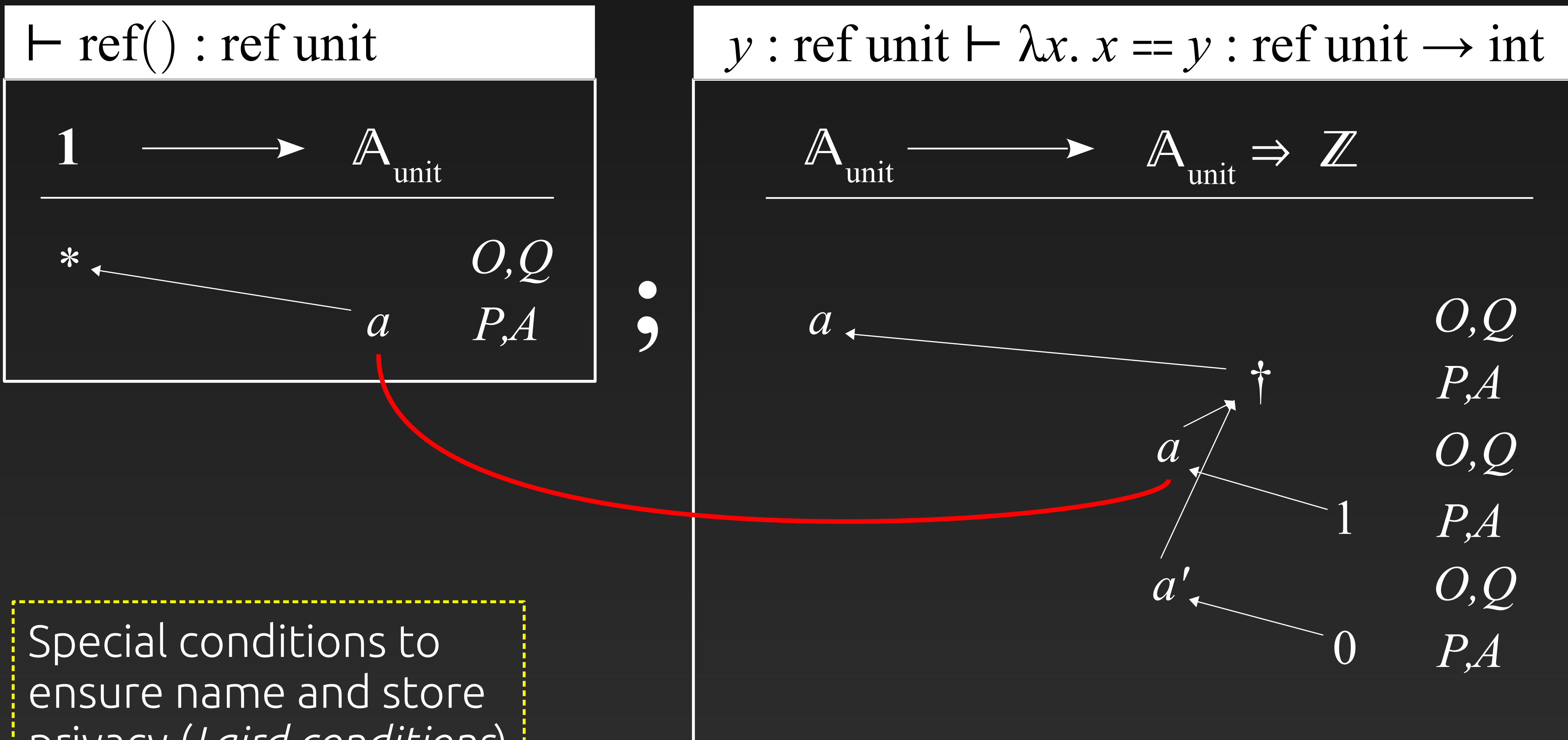


;

$y : \mathrm{ref\ unit} \vdash \lambda x.\, x == y : \mathrm{ref\ unit} \to \mathrm{int}$

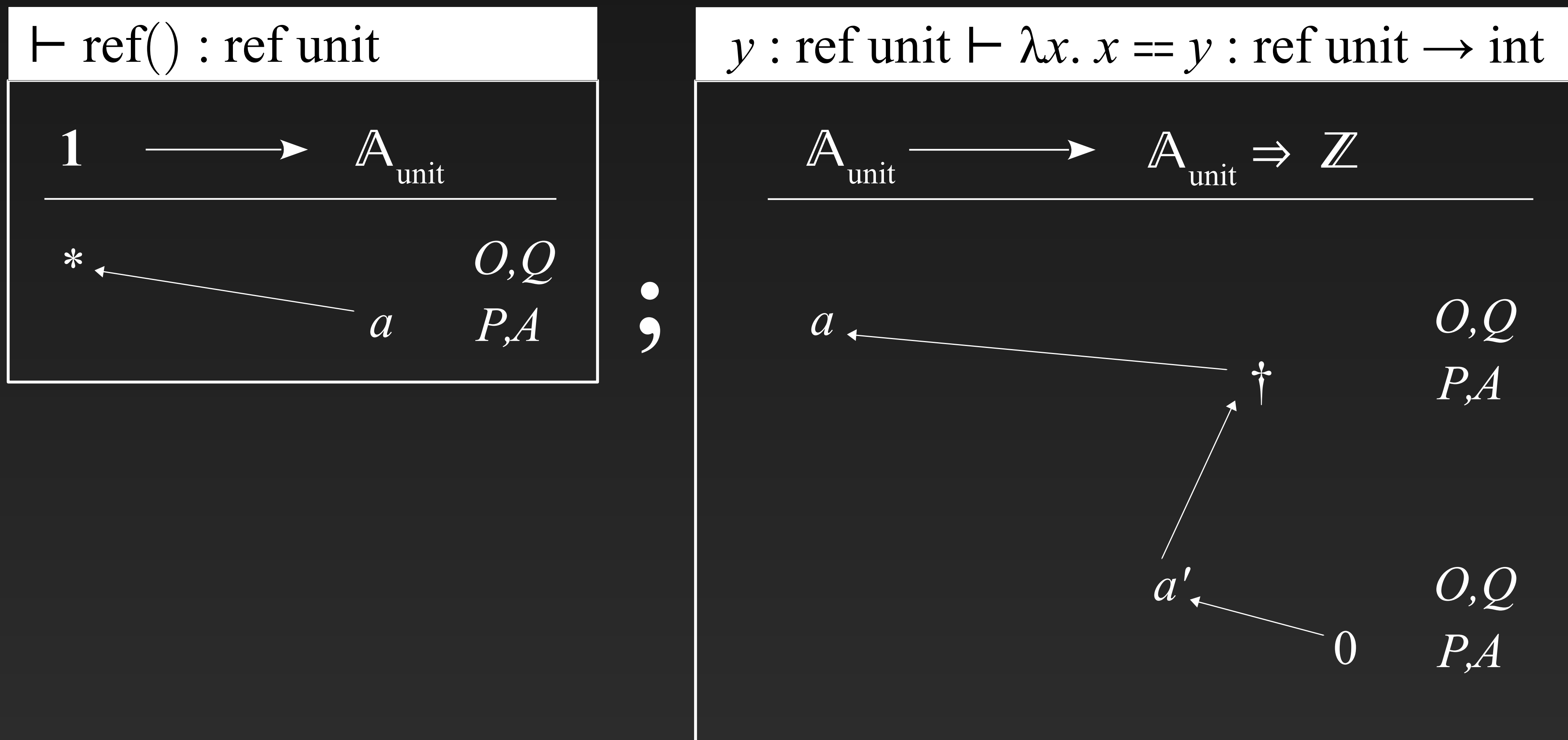


Nominal composition example



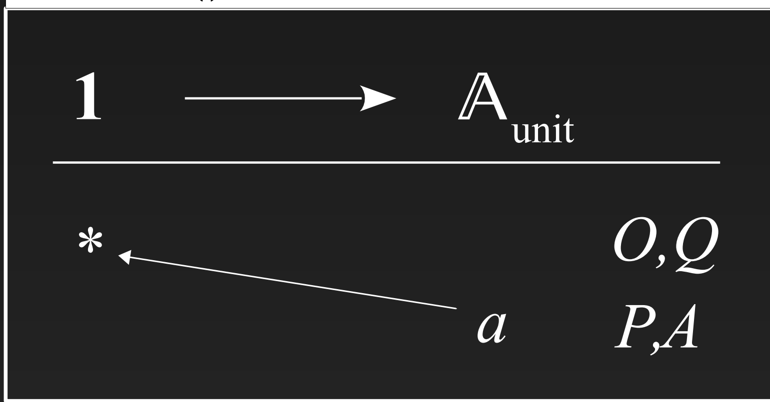
Special conditions to ensure name and store privacy (*Laird conditions*)

Nominal composition example



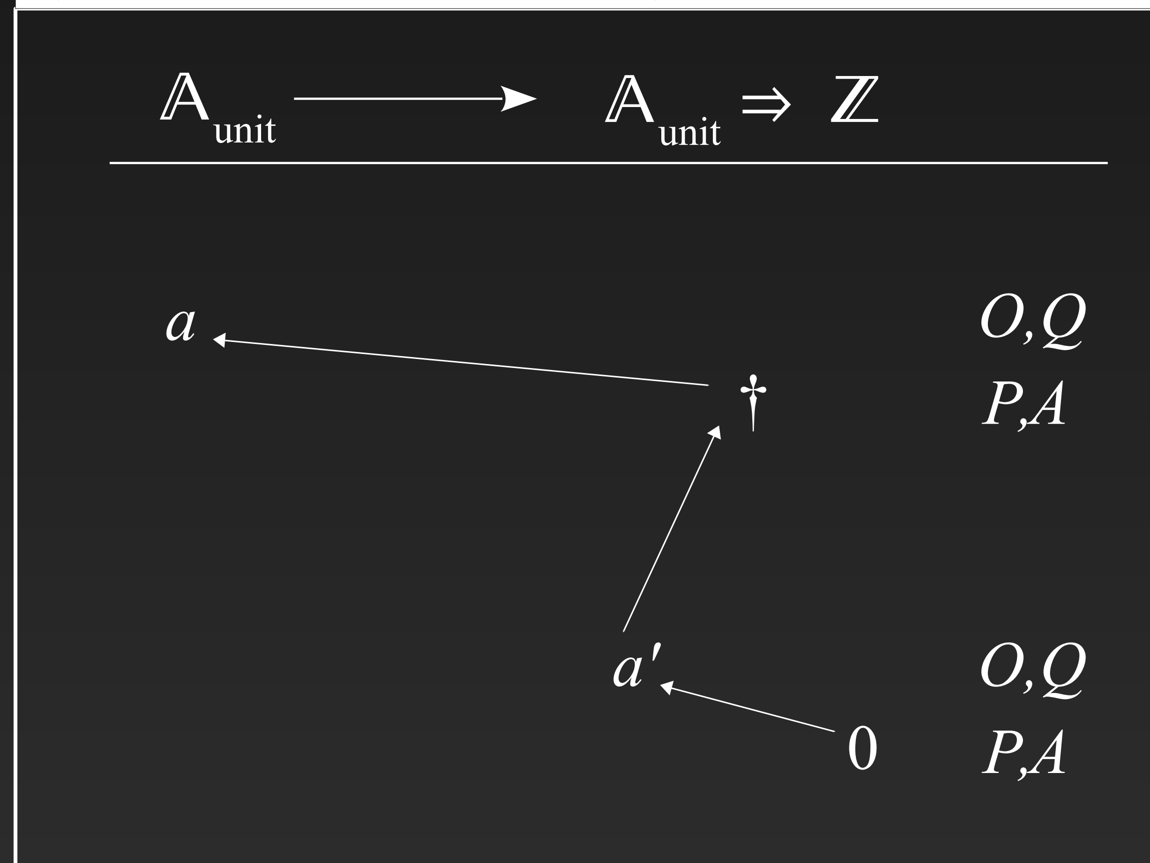
Nominal composition example

$\vdash \mathrm{ref}() : \mathrm{ref\ unit}$



;

$y : \mathrm{ref\ unit} \vdash \lambda x.\, x == y : \mathrm{ref\ unit} \rightarrow \mathrm{int}$



$$= [\![\lambda x.0 : \mathrm{ref\ unit} \rightarrow \mathrm{int}]\!]$$

Taking stock

Games yield denotational models, which:

- are compositional / operational
- precisely capture what is going on:
  - sound & complete: $P \cong P' \iff [\![P]\!] = [\![P']\!]$
  - every "computable" strategy is in the language

Taking stock

Games yield denotational models, which:

- are compositional / operational
- precisely capture what is going on:
  - sound & complete: $P \cong P' \iff [\![P]\!] = [\![P']\!]$
  - every “computable” strategy is in the language

Effects = – conditions + name generators

- Ground store: – innocence + ground reference names
- Higher-order store: – visibility + HO reference names
- Exceptions: – well-bracketing + exception names

Nominal games portfolio

- Full abstraction for languages with:
  - References (ground & higher-order)
  - Concurrency (higher-order channels)
  - Exceptions (local, private)
  - Objects (Interface MJ)

`dblp`: Laird, Murawski, T.

- Trace models

`dblp`: Laird, Ghica, Jaber, T.

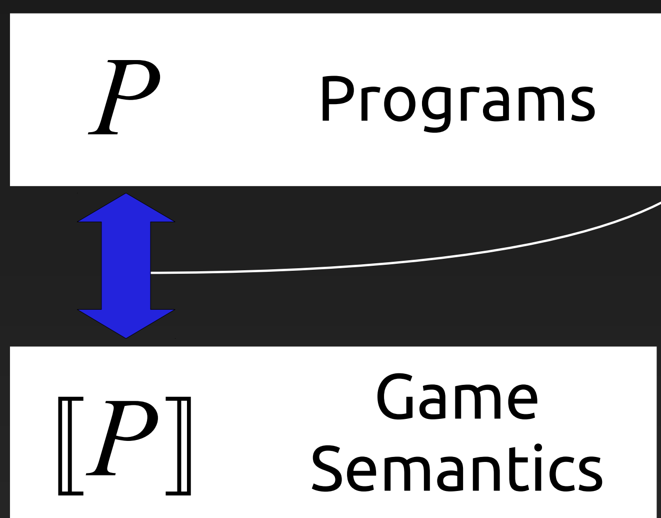
- Algorithmic games

Algorithmic game semantics

P	Programs
$\llbracket P \rrbracket$	Game Semantics

$$P \cong P' \Leftrightarrow \llbracket P \rrbracket = \llbracket P' \rrbracket$$

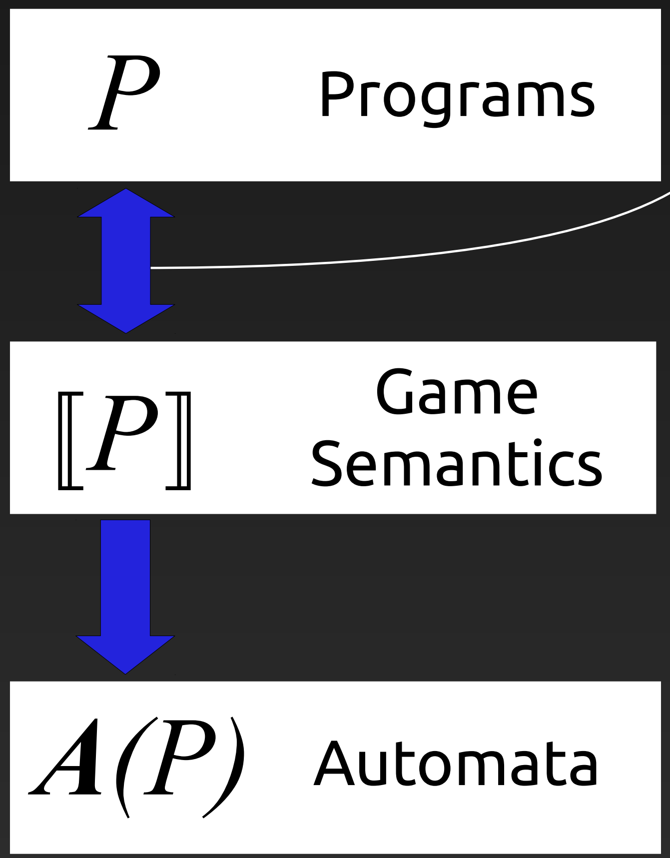
Algorithmic game semantics



$$[\![\lambda x.\ \mathrm{ref}()]\!] = \{\ *\ \dagger\ *\ a_1\ *\ a_2\ \ldots\ \}$$

$$P \cong P' \Leftrightarrow [\![P]\!] = [\![P']\!]$$

Algorithmic game semantics



$$[\![\lambda x.\ \mathrm{ref}()]\!] = \{\ *\ \dagger\ *\ a_1\ *\ a_2\ \ldots\ \}$$

$$P \cong P' \Leftrightarrow [\![P]\!] = [\![P']\!]$$

$$P \vdash \varphi \ \Leftrightarrow\ A(P) \vdash \varphi$$

Nominal Automata

The need for nominal automata

- Nice (fully abstract) game models
- Represented as sets of strings (with structure)

- *How to make them algorithmic/automated?*
- Use automata

- *But, the alphabet is infinite...*
- Use nominal automata (i.e. automata over infinite alphabets)

Fresh-register automata

$$[\![\lambda x.\ \mathrm{ref}()]\!] = \{ * \dagger * a_1 * a_2 \ldots \mid a_i\text{'s distinct} \}$$

Automata with names

- Infinite alphabet
- Freshness recognition

Fresh-register automata

$$[\![\lambda x.\ \mathrm{ref}()]\!] = \{\ * \dagger * a_1 * a_2 \ldots \mid a_i\text{'s distinct}\ \}$$

Automata with names

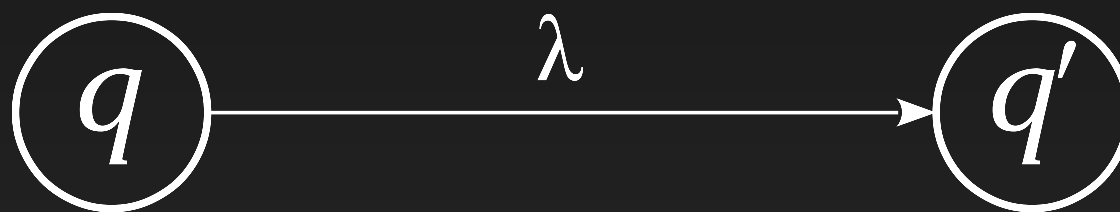
- Infinite alphabet
- Freshness recognition

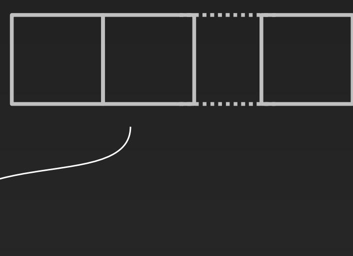
Part of a huge body of work on automata over infinite alphabets!

Finite-state machines with registers

Register Automata (RA)

Let $\Sigma = \{a_1, a_2, \ldots, a_n, \ldots\}$ be an infinite alphabet of names





finitely many (say R) registers

registers store names

Label λ of the form:

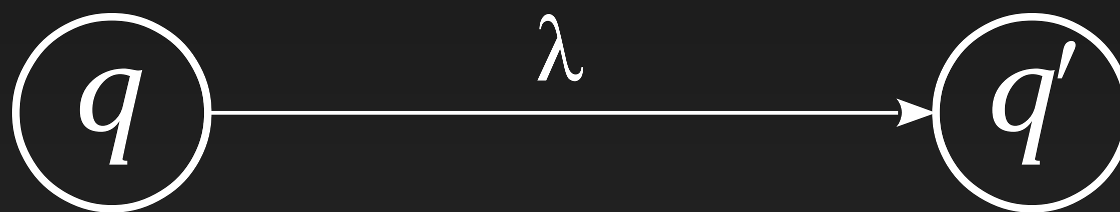
- read(i), $i \in \{1,\ldots,R\}$
- fresh(i), $i \in \{1,\ldots,R\}$
- κ, $\kappa \in F$

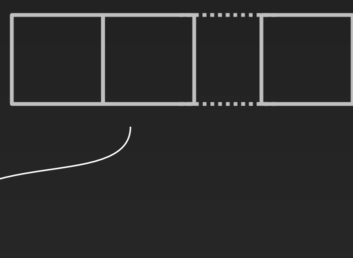
a finite set of constants

[Kaminski & Fraencez '94]

Register Automata (RA)

Let $\Sigma = \{a_1, a_2, \ldots, a_n, \ldots\}$ be an infinite alphabet of names





finitely many (say R) registers

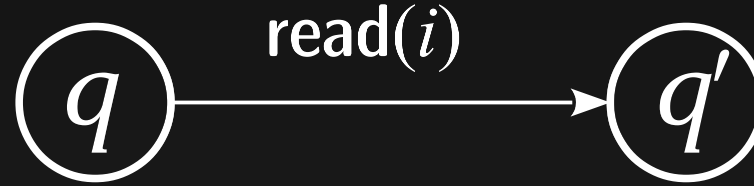
registers store names

Label λ of the form:

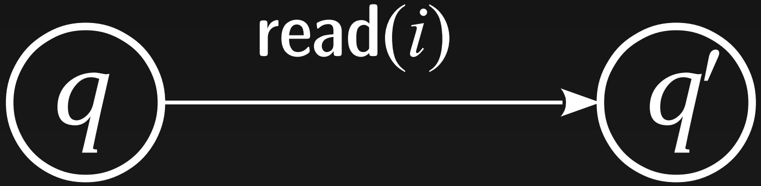
- **read**(i), $i \in \{1,\ldots,R\}$
- **fresh**(i), $i \in \{1,\ldots,R\}$
- κ, $\kappa \in F$

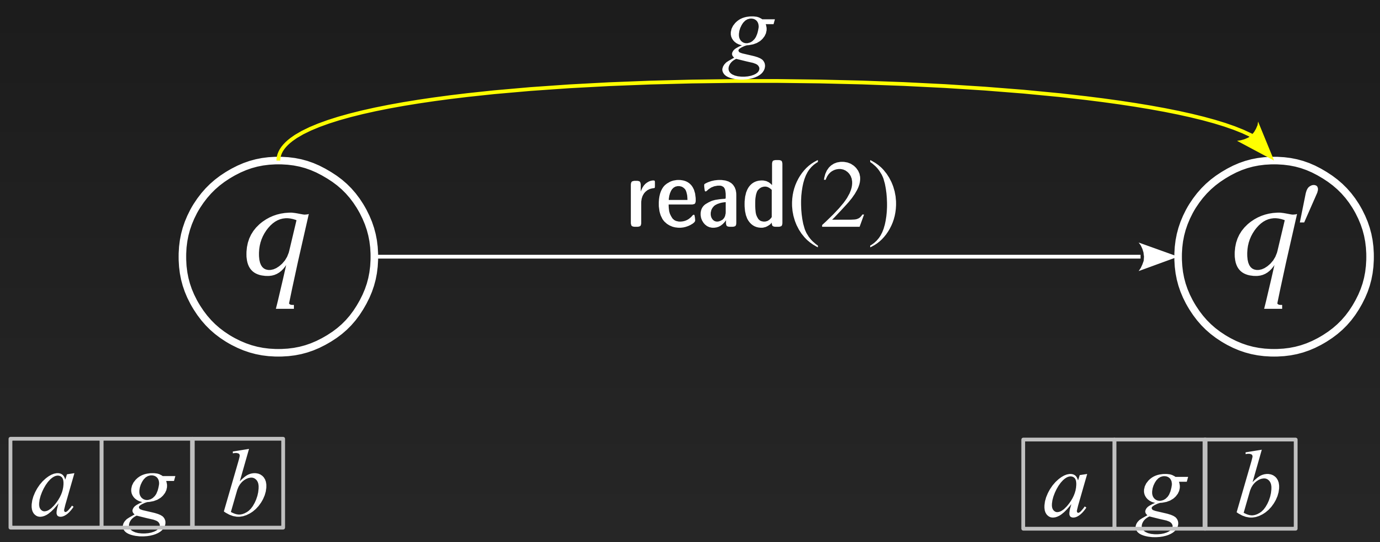
a finite set of constants

Transitions:

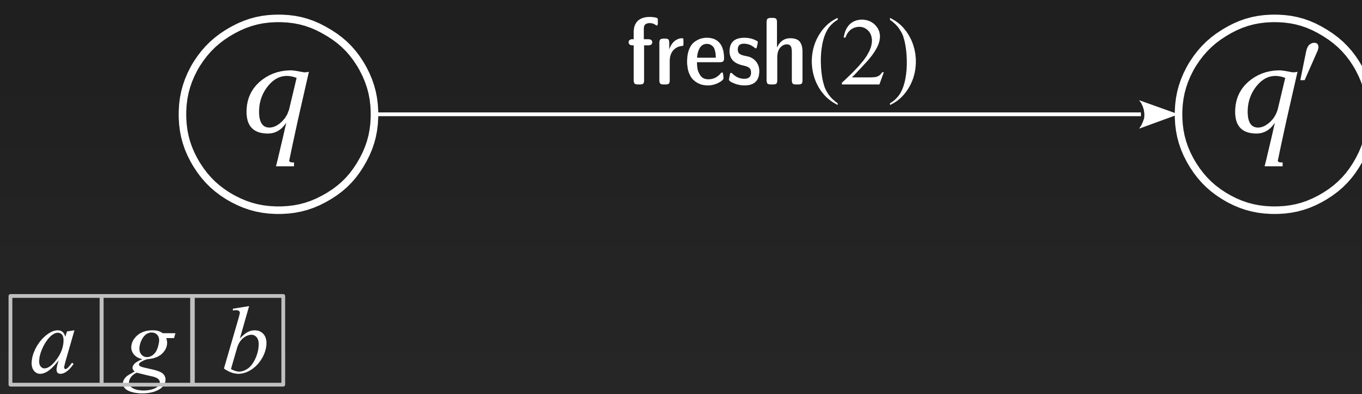


Transitions:

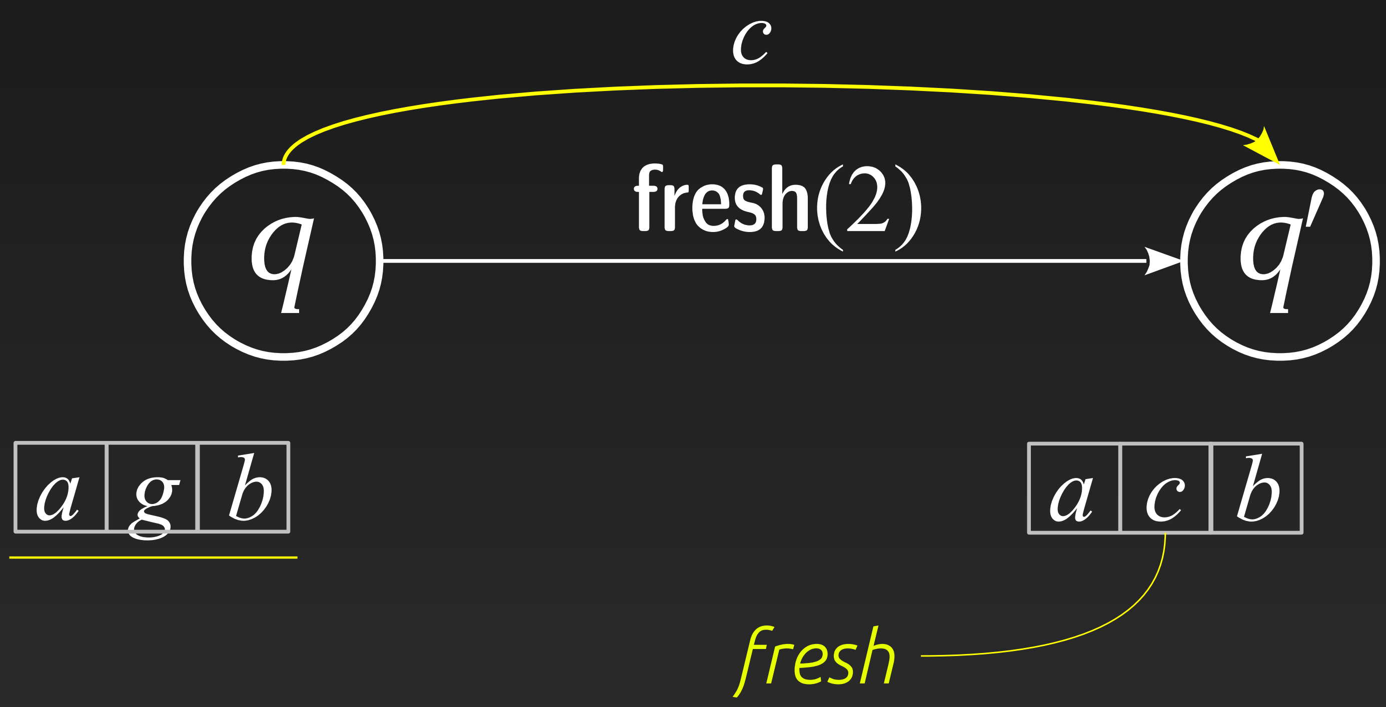




Transitions:



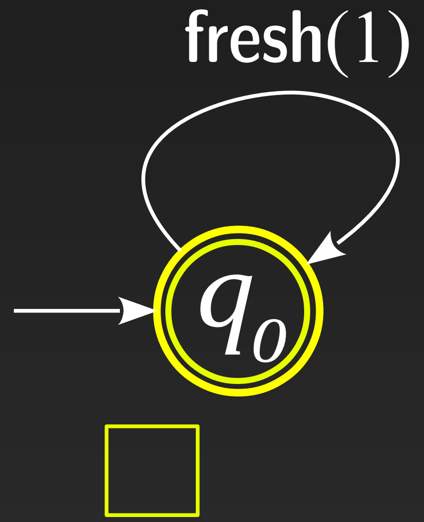
Transitions:



Example

$$L_1 = \{ a_1 a_2 ... a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

(all strings where each name is distinct from its predecessor)



Example

$$L_1 = \{\, a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1} \,\}$$

(all strings where each name is distinct from its predecessor)

a

Example

$$L_1 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid\ n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)

$a\,b$

Example

$$L_1 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid\ n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)

$a\ b\ c$

Example

$$L_1 = \{\, a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1} \,\}$$

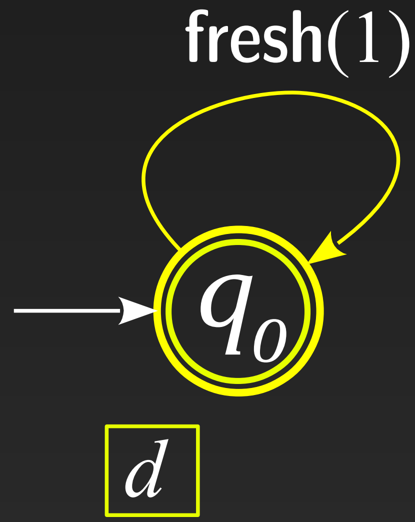
(all strings where each name is distinct from its predecessor)

$a\ b\ c\ a$

Example

$$L_1 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid\ n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)

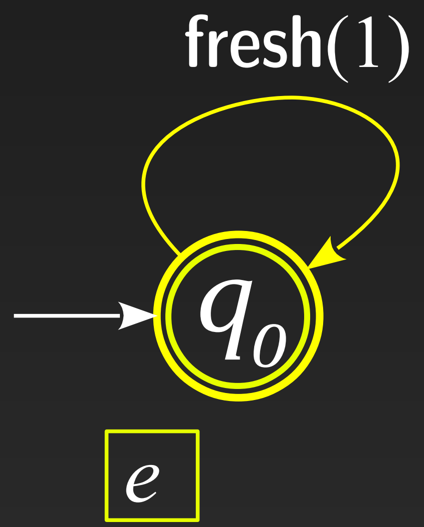


$a\ b\ c\ a\ d$

Example

$$L_1 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid\ n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1}\ \}$$

(all strings where each name is distinct from its predecessor)

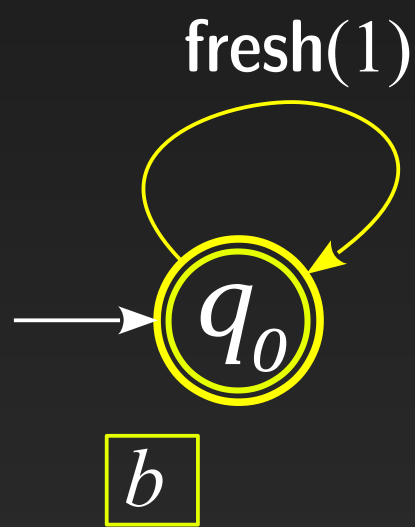


$a\ b\ c\ a\ d\ e$

Example

$$L_1 = \{\, a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1} \,\}$$

(all strings where each name is distinct from its predecessor)



$a\ b\ c\ a\ d\ e\ b$

Example

$$L_1 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i < n.\ a_i \neq a_{i+1} \}$$

(all strings where each name is distinct from its predecessor)

$a\ b\ c\ a\ d\ e\ b\ a\ g\ c\ a\ b\ a\ b$

Example

$$L_1 = \{\, a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0,\ \forall i < n.\ a_i \neq a_{i+1} \,\}$$

(all strings where each name is distinct from its predecessor)

$a\ b\ c\ a\ d\ e\ b\ a\ g\ c\ a\ b\ a\ b\ a\ n\ d\ w\ e\ l\ o\ v\ e\ M\ S\ R$

Global freshness?

```
public void foo() {
  // Create new list
  List x = new ArrayList();

  x.add(1); x.add(2);
  Iterator i = x.iterator();
  Iterator j = x.iterator();
  i.next(); i.remove(); j.next();
}
```

we can express safety properties:

if an iterator modifies its collection x *then other iterators of* x *become invalid*

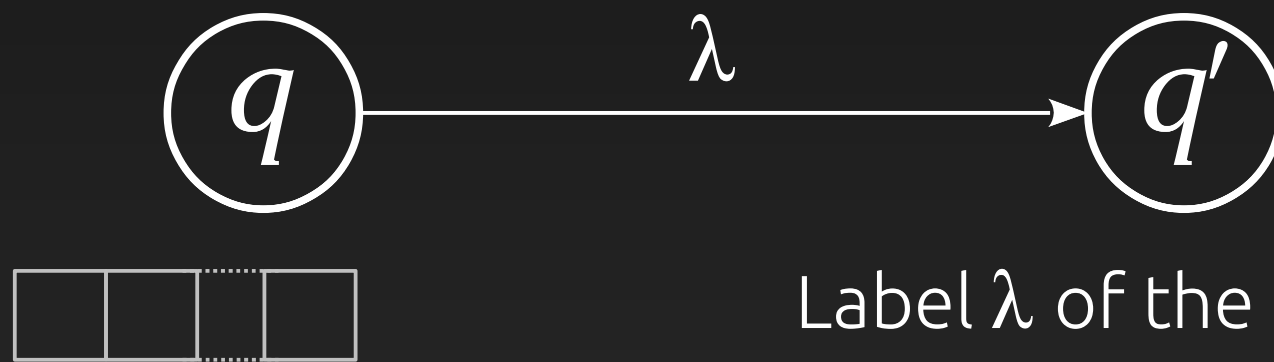
(the code on the left is bad)

[Grigore, Distefano, Petersen & T. '13]

but we cannot model **new**

Fresh-Register Automata (FRA)

Let $\Sigma = \{a_1, a_2, \ldots, a_n, \ldots\}$ be an infinite alphabet of names



finitely many (say R) registers

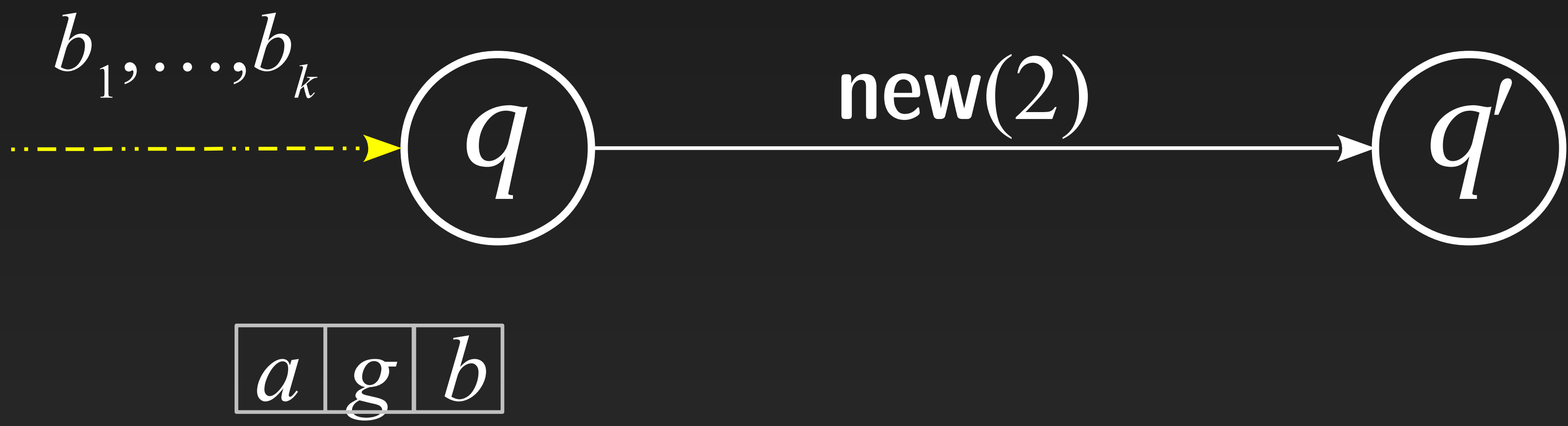
registers store names

Label λ of the form:

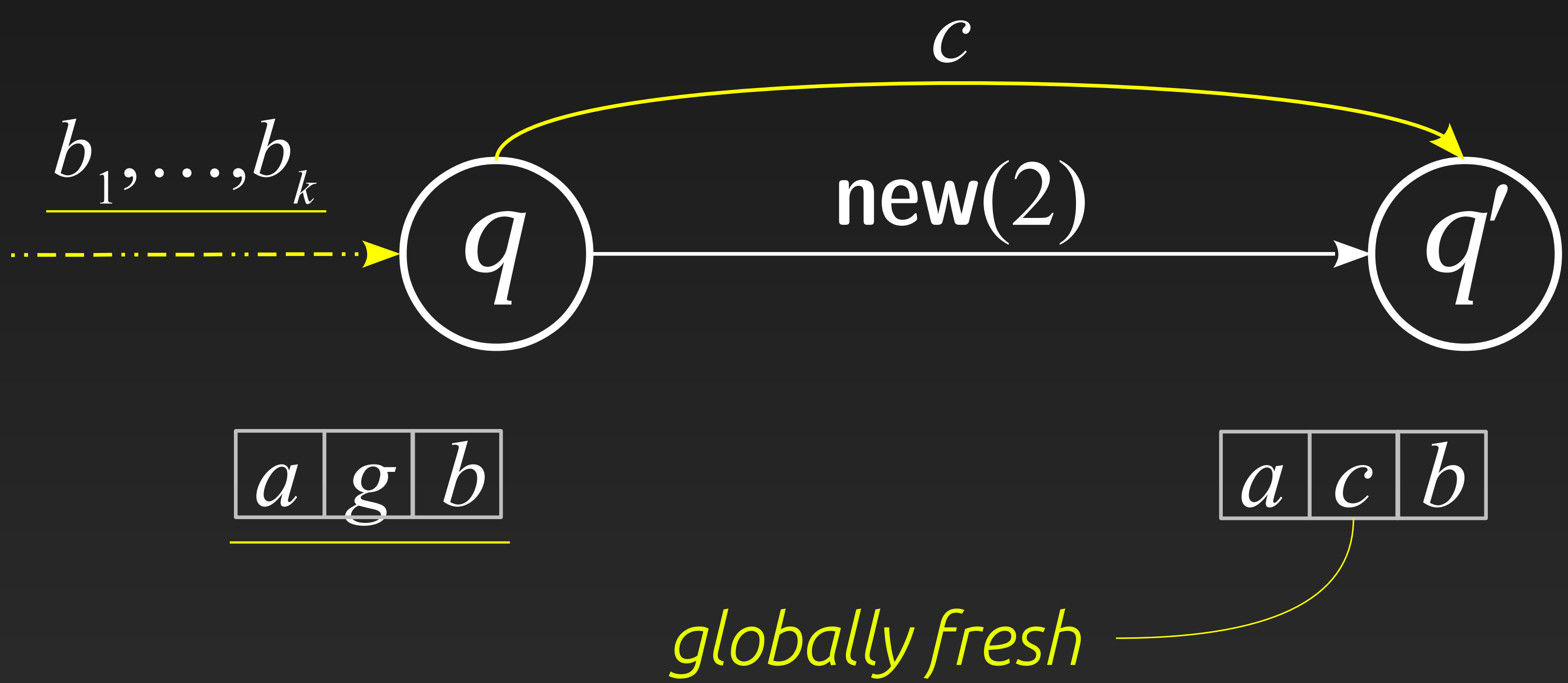
- **read**(i), $i \in \{1,\ldots,R\}$
- **fresh**(i), $i \in \{1,\ldots,R\}$
- κ, $\kappa \in F$
- **new**(i), $i \in \{1,\ldots,R\}$

global freshness oracle

Transitions:



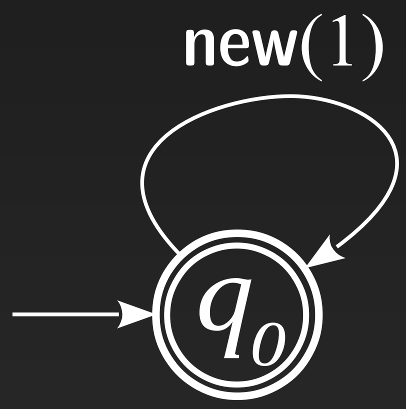
Transitions:



Examples

$$L_2 = \{ a_1 a_2 \ldots a_n \in \Sigma^* \mid n \geq 0, \forall i \neq j.\ a_i \neq a_j \}$$

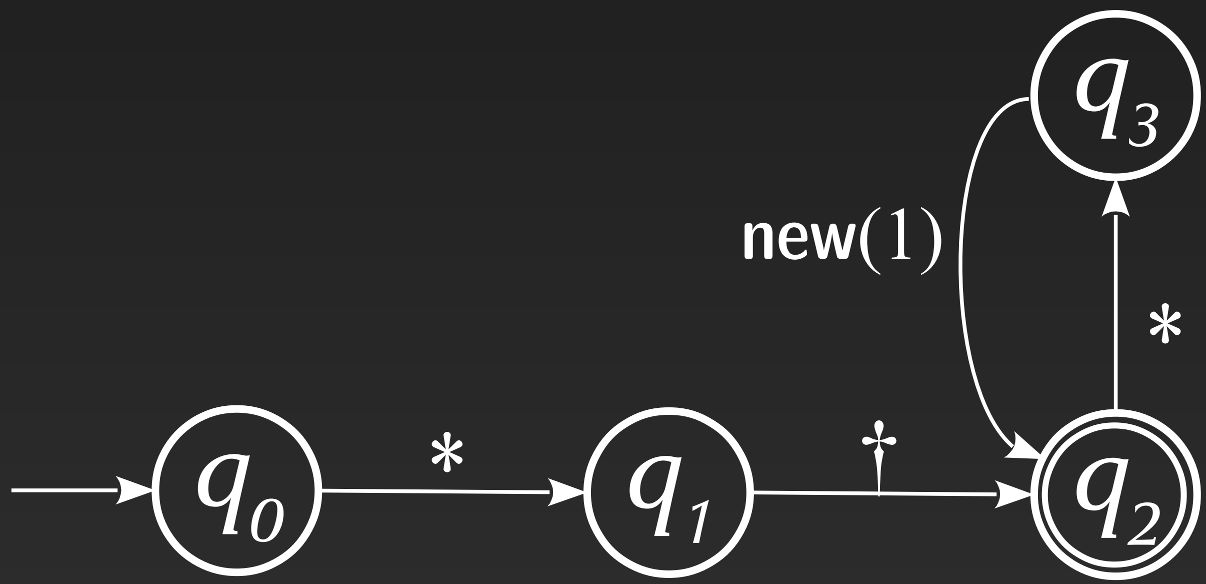
(all strings of pairwise distinct names)



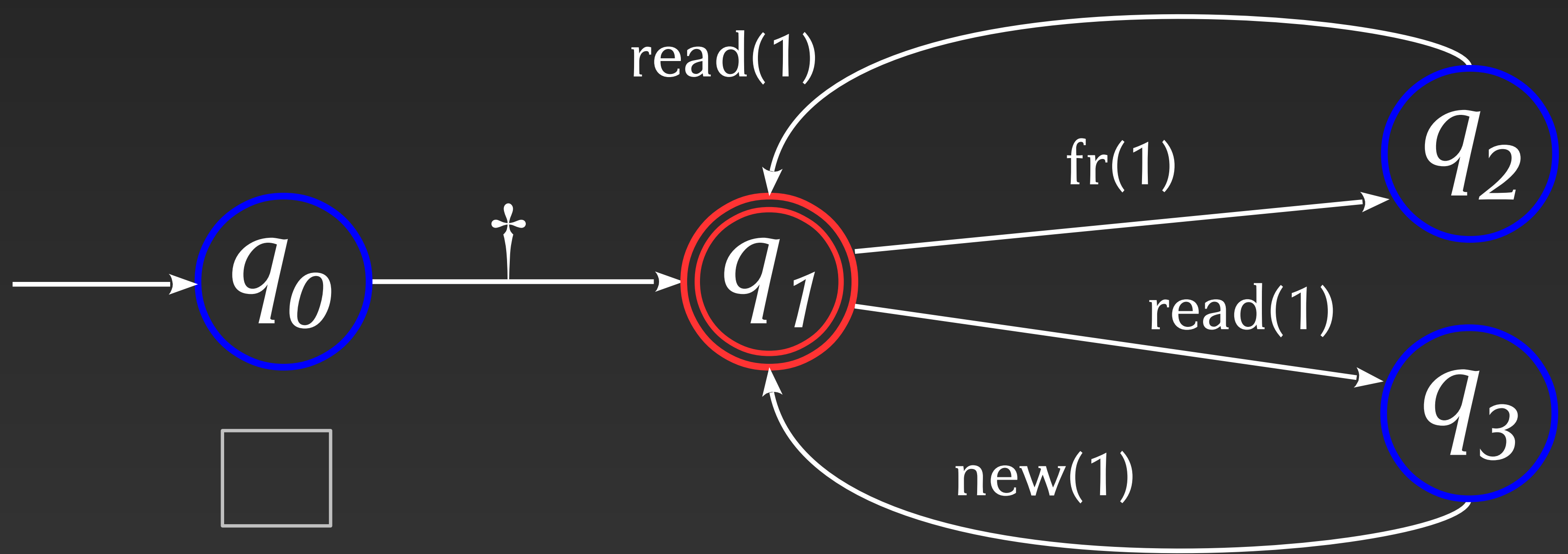
Examples

$$L_2 = \{\ a_1 a_2 \ldots a_n \in \Sigma^* \mid\ n \geq 0, \forall i \neq j.\ a_i \neq a_j \}$$

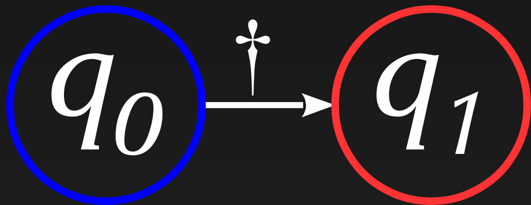
(all strings of pairwise distinct names)

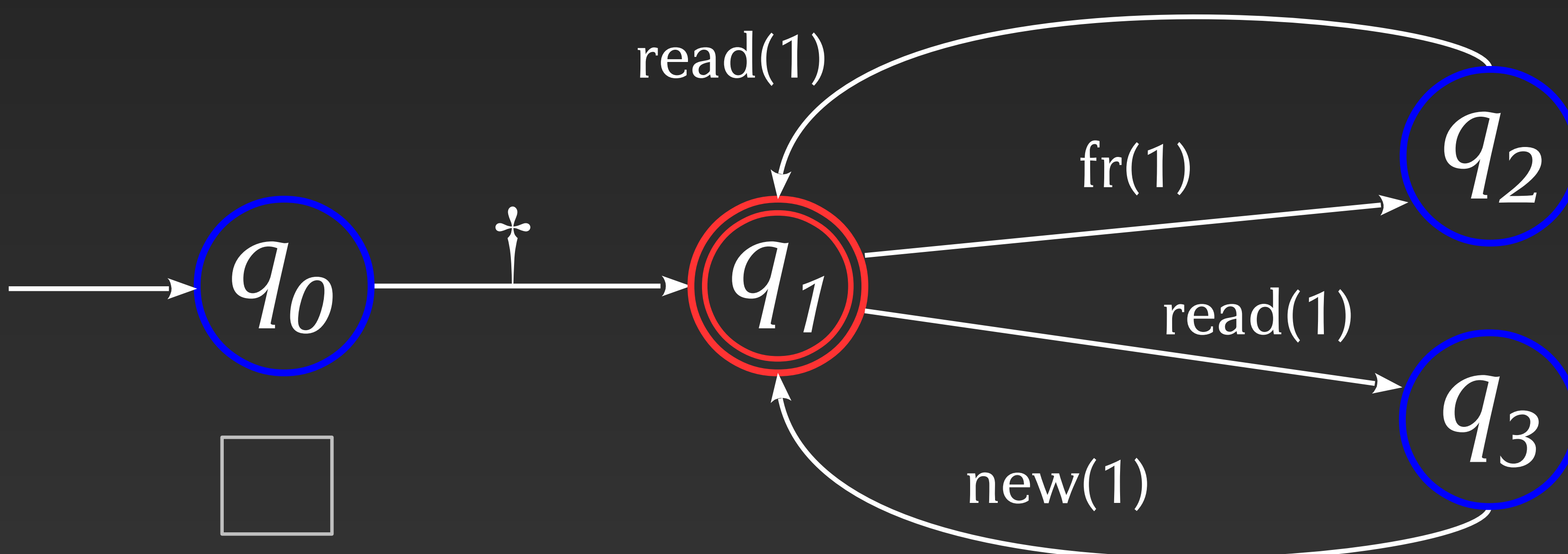


$$[\![\lambda x.\ \mathrm{ref}()]\!] = \{\ * \dagger * a_1 * a_2 \ldots \mid\ a_i\text{'s distinct }\}$$

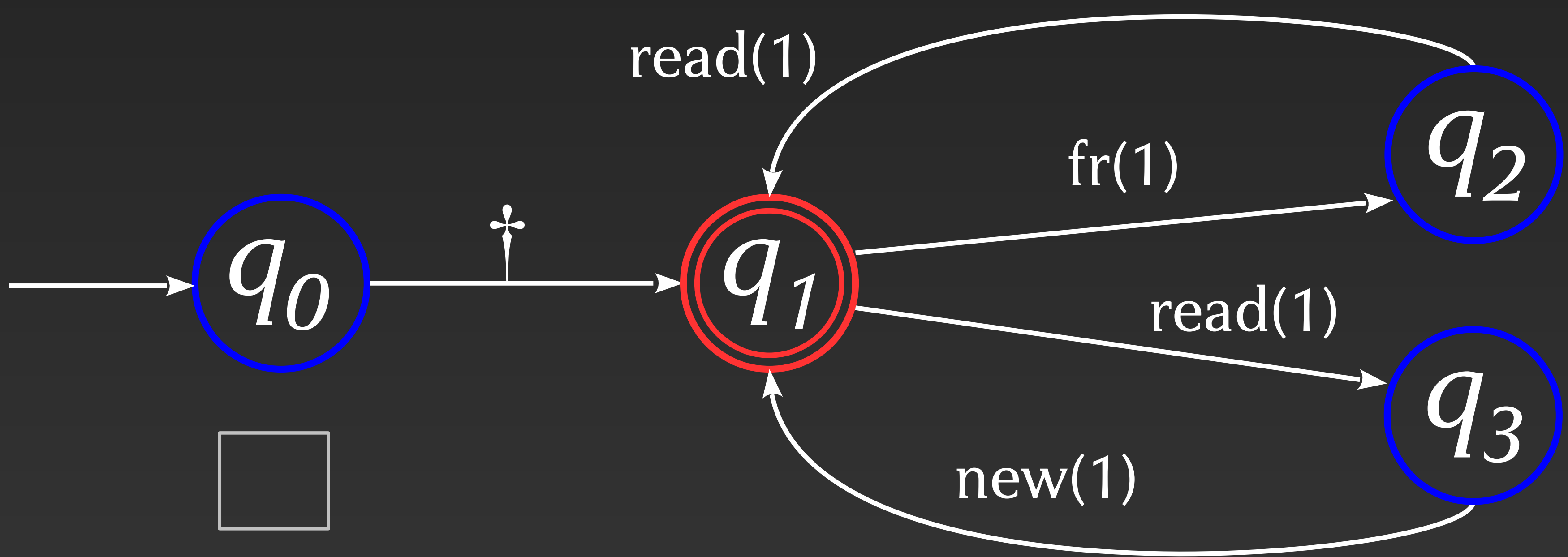


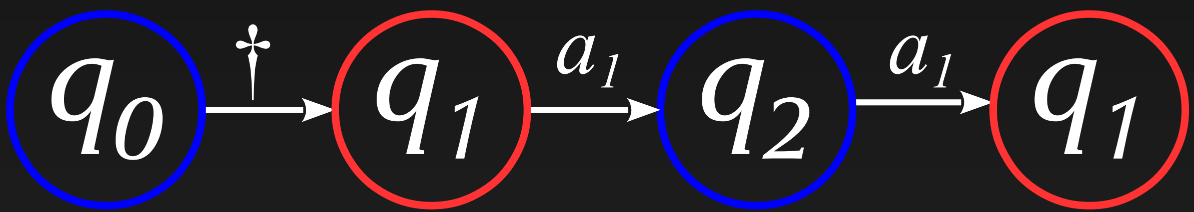
read(1)
fr(1)
read(1)
new(1)
q_0
q_1
q_2
q_3


q_0
†
q_1

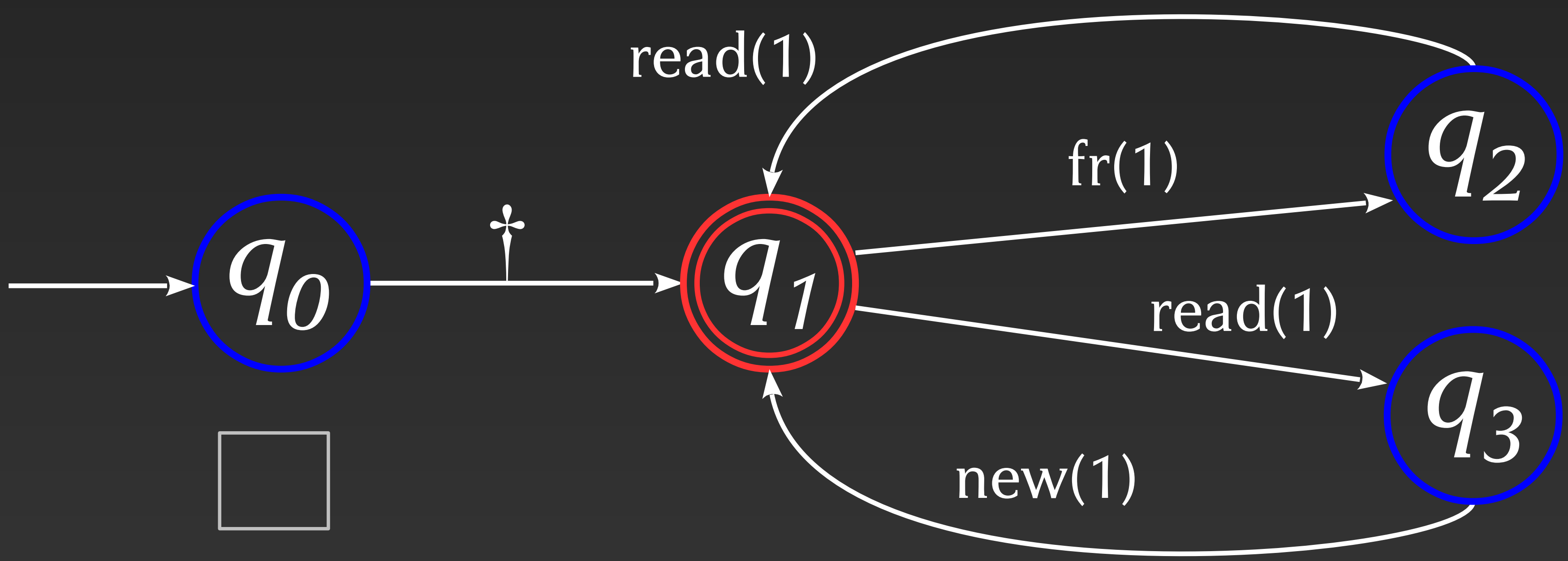

read(1)
fr(1)
read(1)
new(1)
q_0
†
q_1
q_2
q_3


q_0
†
q_1
a_1
q_2

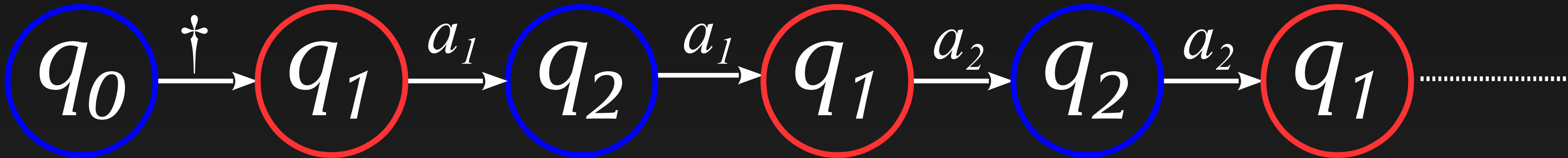

read(1)
fr(1)
read(1)
new(1)
†
q_0
q_1
q_2
q_3



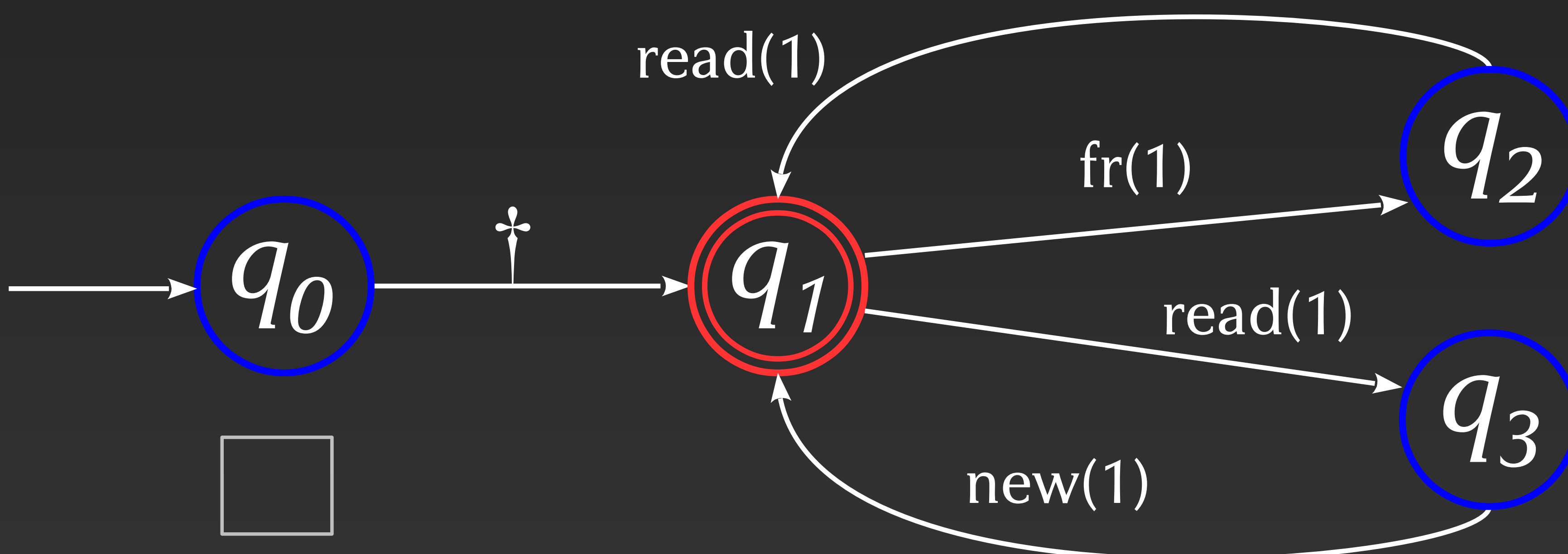
q_0
†
q_1
a_1
q_2
a_1
q_1



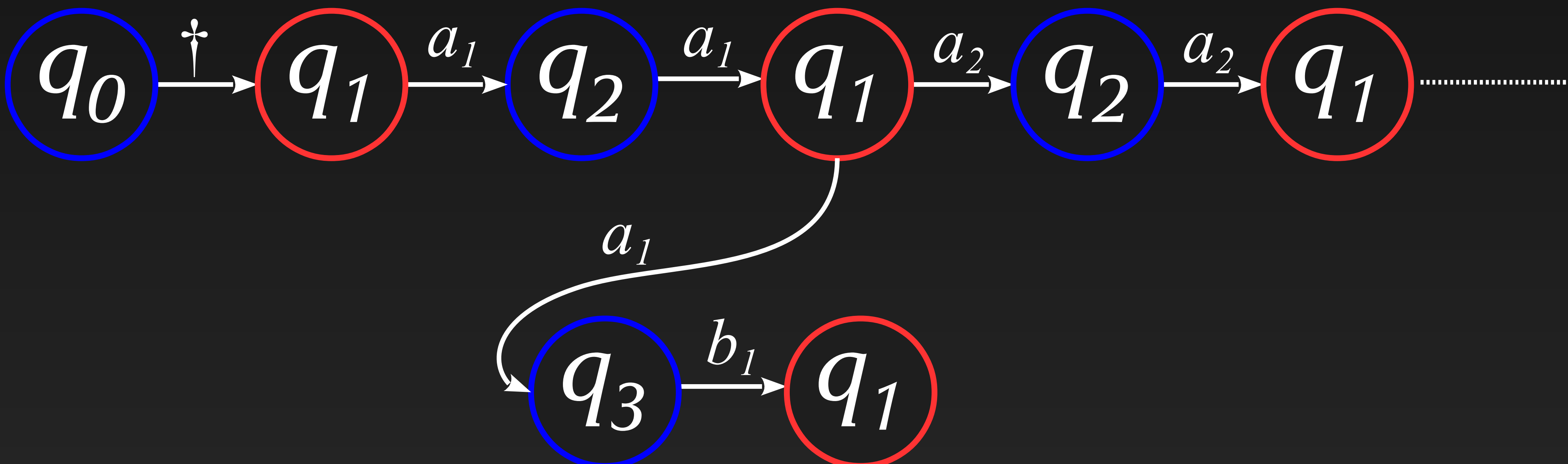
read(1)
fr(1)
q_2
q_0
†
q_1
read(1)
q_3
new(1)

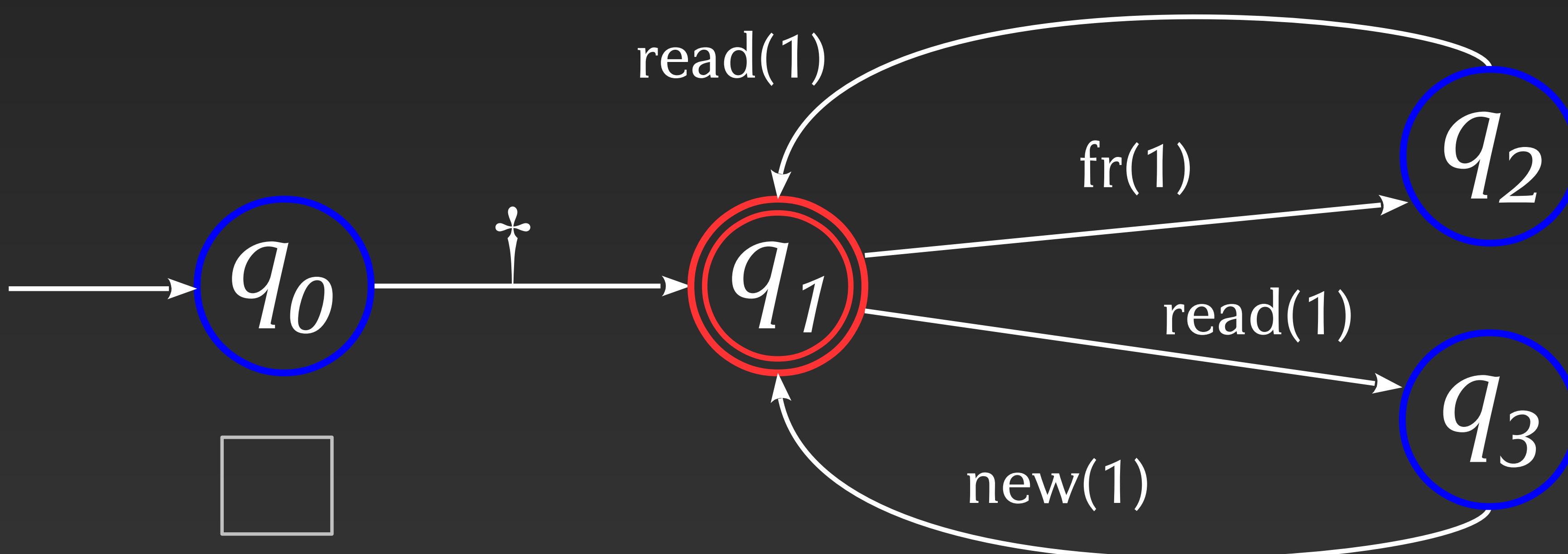


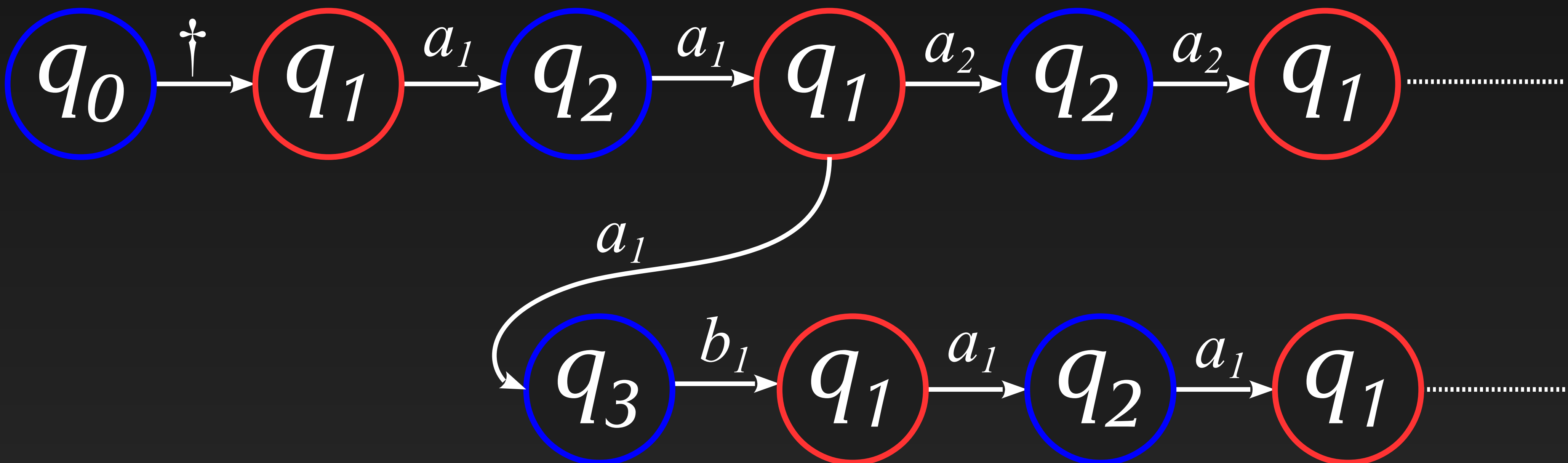
q_0
†
q_1
a_1
q_2
a_1
q_1
a_2
q_2
a_2
q_1

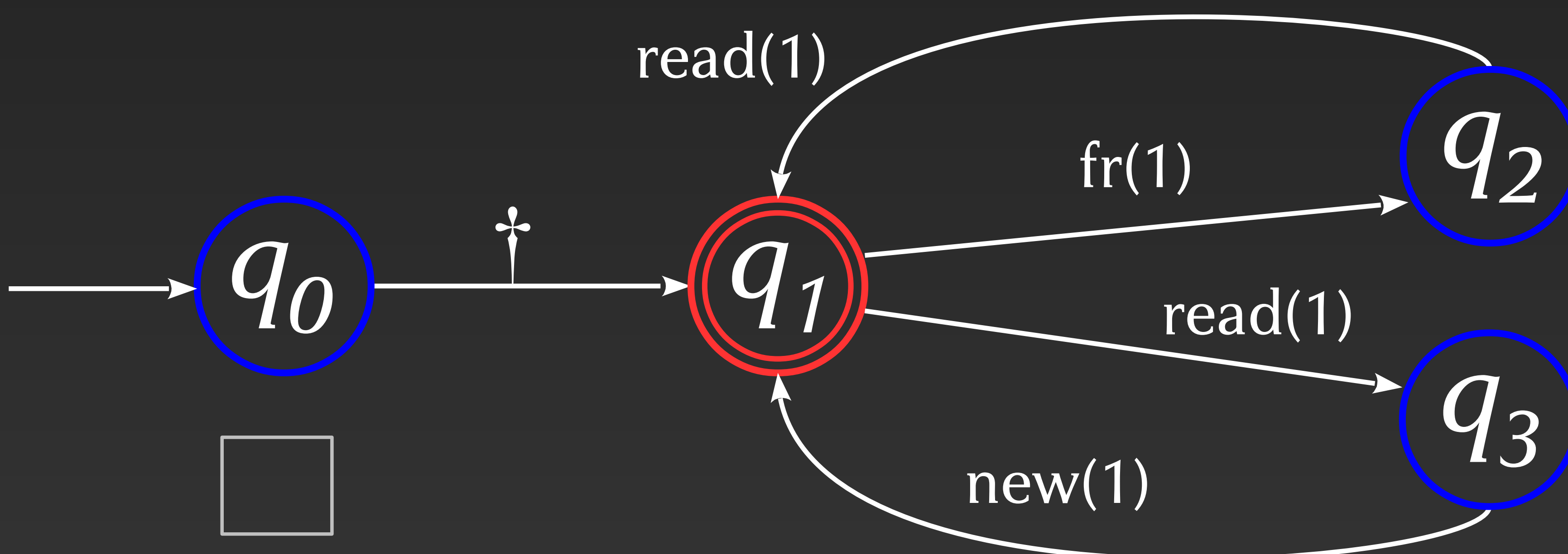


read(1)
fr(1)
q_2
q_0
†
q_1
read(1)
q_3
new(1)

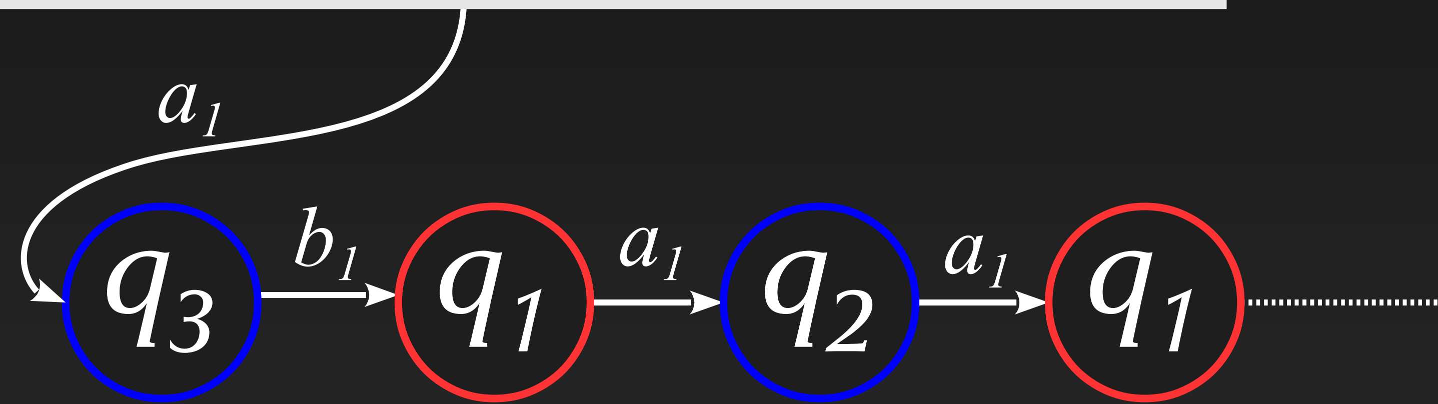

q_0
†
q_1
a_1
q_2
a_1
q_1
a_2
q_2
a_2
q_1
a_1
q_3
b_1
q_1

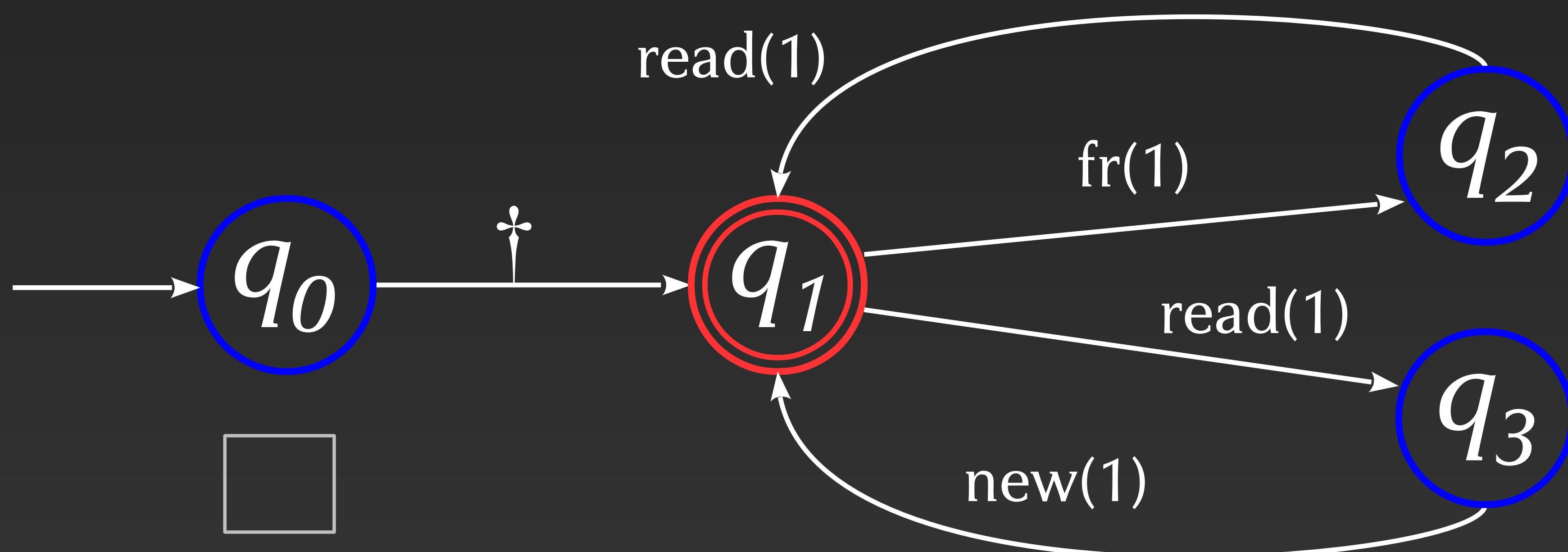

read(1)
q_2
fr(1)
q_0
†
q_1
read(1)
q_3
new(1)


q0
†
q1
a1
q2
a1
q1
a2
q2
a2
q1
a1
q3
b1
q1
a1
q2
a1
q1


read(1)
fr(1)
q2
q0
†
q1
read(1)
q3
new(1)

```
let y = ref(ref()) in
    λx. y := ( if x==!y then ref() else x ); !y
    : unit ref → unit ref
```





Properties of FRAs

- Cleanly extend Register Automata (RA's)
- Closed under union and intersection;
  not under complementation, concatenation, _*
- Universality undecidable (from RA's)
- Emptiness decidable (from RA's)
- Bisimilarity decidable

Algorithmic nominal game semantics

$$P \cong P' \iff A(P) \sim A(P')$$

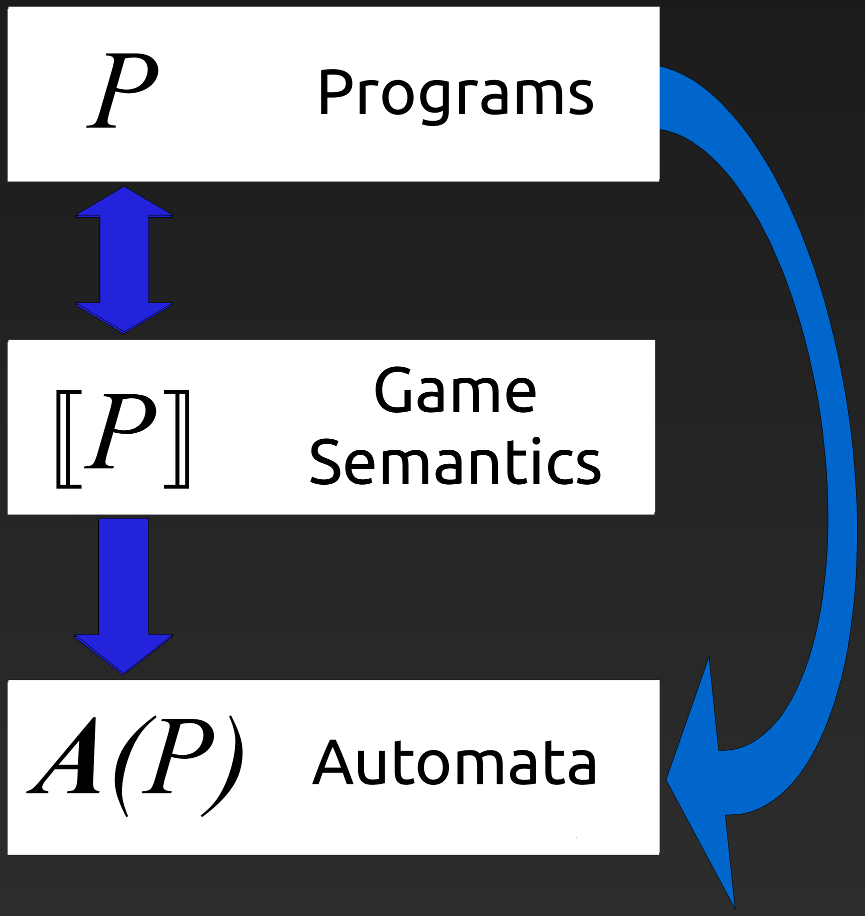
Representation of strategies as FRA's

- with “data words”
- pushdown variants

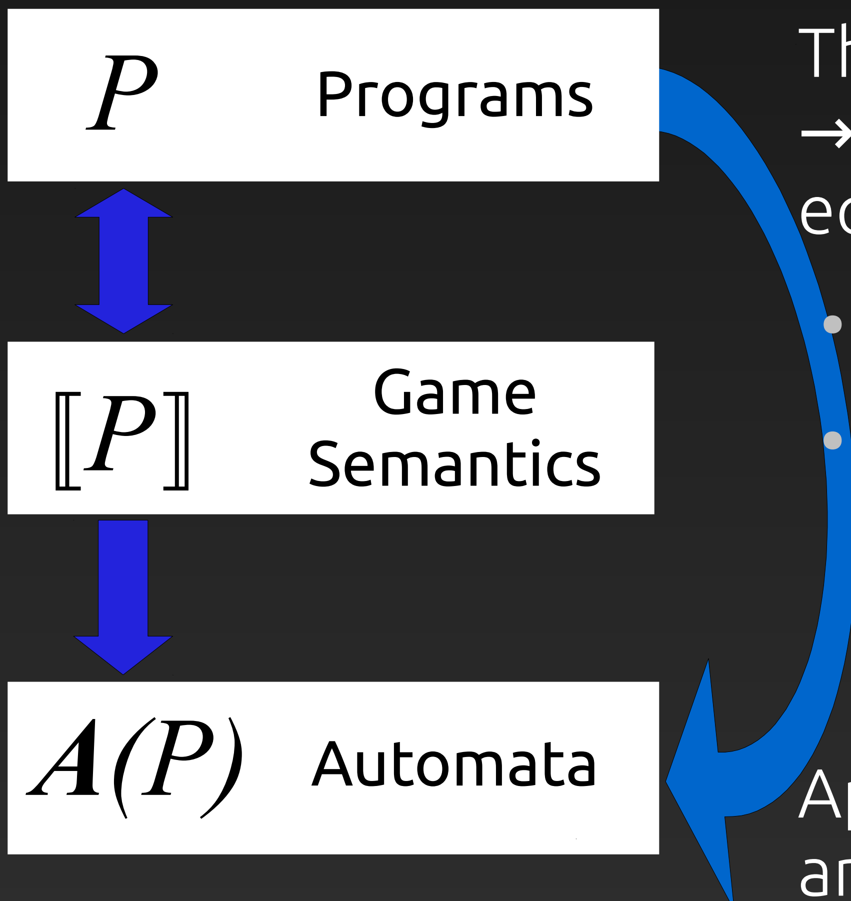
pairs (κ, a):
- κ from finite alph.
- a a name

stack stores constants and register “snapshots”

From programs to automata



From programs to automata



The passage can be done automatically
→ decision procedure for program equivalence

- Possible at specific types
- Full type-based classification
(at each type, either the problem is undecidable or we get a procedure)

Applied to ML with ground references and Interface Middleweight Java

[Murawski & T. '11,'12; Murawski, Ramsay & T.]

Nominal automata in program verification

- Algorithmic game semantics
- Pushdown analyses (pushdown FRA's!)
  - Pointers, Concurrent processes
  - Reachability, MSO-decidability

`dblp`: Atig, Bouajjani, Fratani & Qadeer ; Bollig *etal*

- Runtime Java verification
  - TOPL (register automata)

[Grigore, Distefano, Petersen & T. '13]

Summary and what's next

Our journey: programs → games → automata

- games capture essential program behaviour
- denotational / operational / algorithmic

Further on:

- More effects (can we have one model for all?)
- Verification
  - Expand algorithmic approach to gen. model checking
  - Program logics and games
- Games as a paradigm for open/HO programs

Summary and what's next

thanks

Our journey: programs → games → automata

- games capture essential program behaviour
- denotational / operational / algorithmic

Further on:

- More effects (can we have one model for all?)
- Verification
  - Expand algorithmic approach to gen. model checking
  - Program logics and games
- Games as a paradigm for open/HO programs