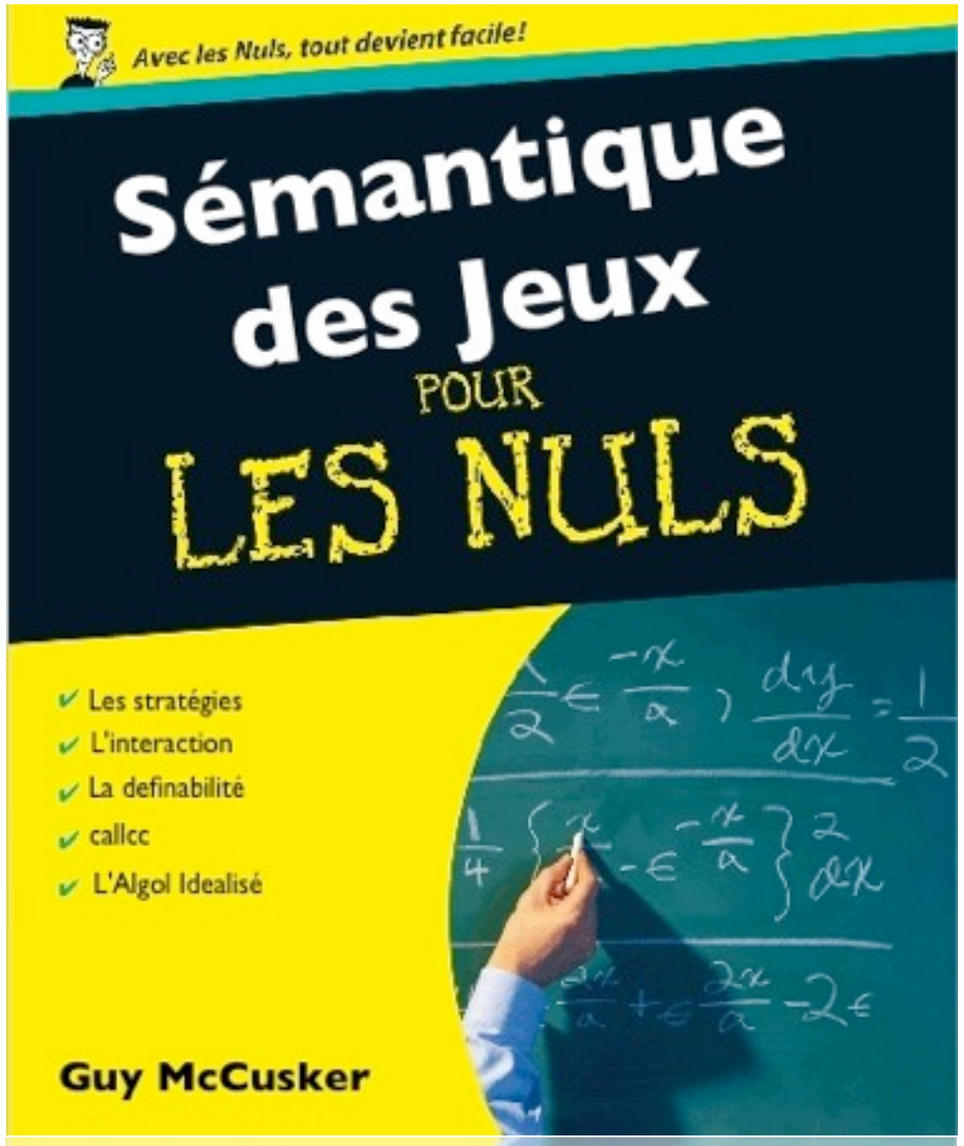
Programs and Strategies

Guy McCusker

Logic and Interaction 2012

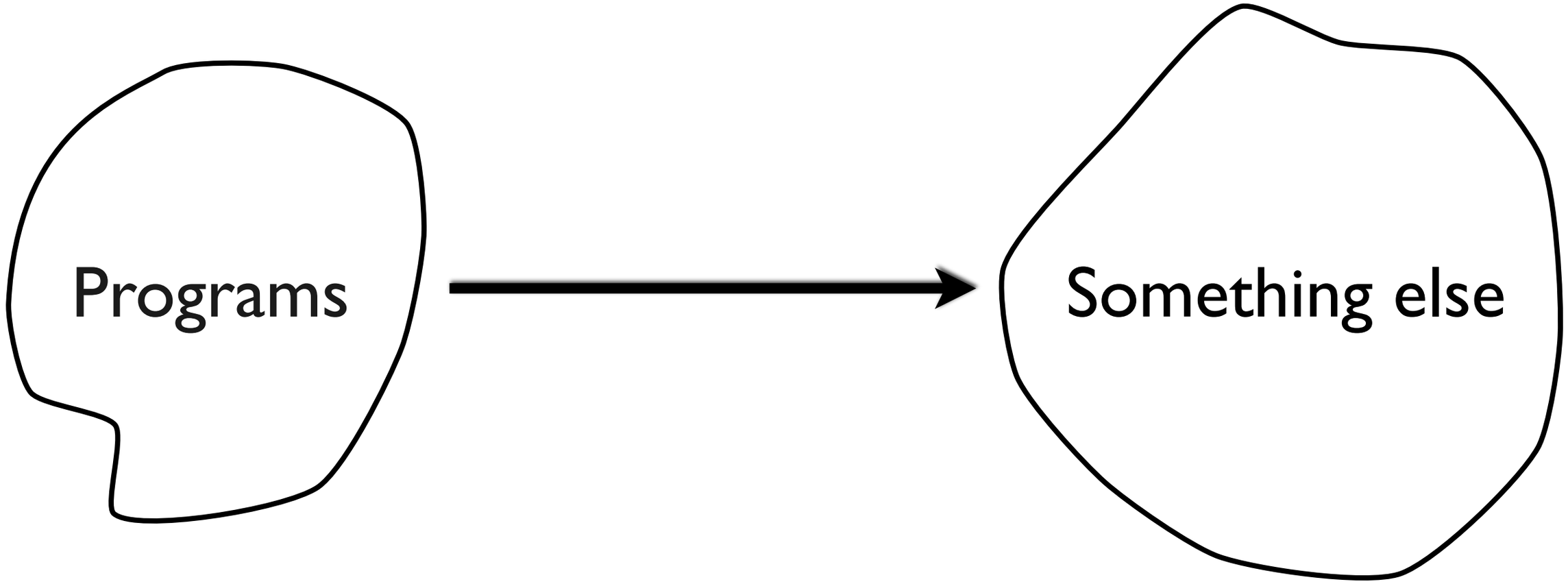


Alternative title

Game semantics for programs

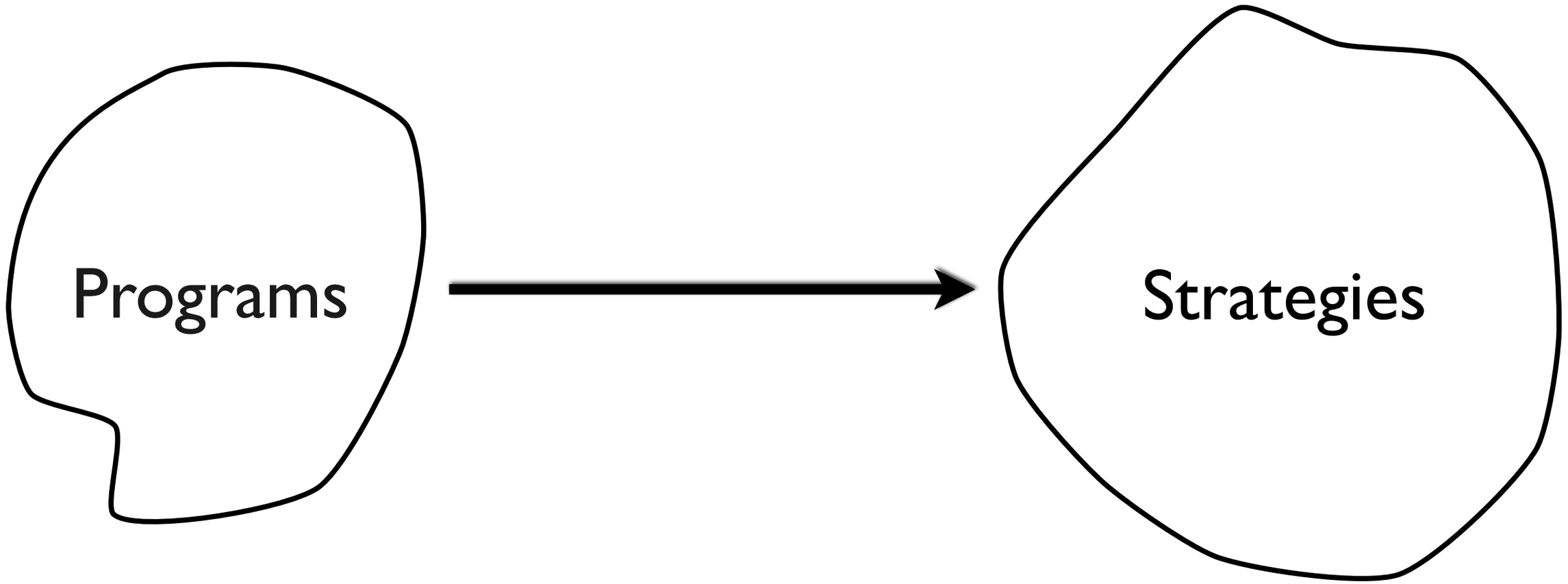
- Game semantics models *programs* as certain kinds of *strategies* on games.
- Different sorts of programs give rise to different sorts of strategies.
- Properties of strategies give an *abstract characterization* of computational behaviour.

The semantics game



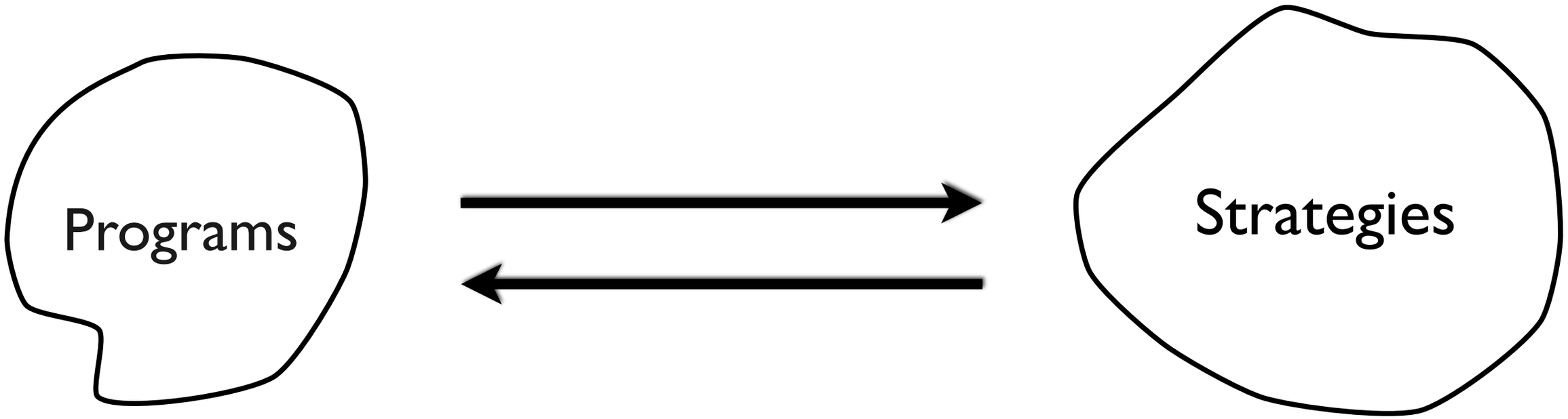
Every program corresponds to a something

Game Semantics



Every program corresponds to a strategy

Definability



Every program corresponds to a strategy

... and every (finite) strategy comes from a program

Game Semantics

- We will see how game semantics gives models with definability for:
  - pure functional programs
  - imperative programs
  - programs with control operators
  - programs with higher-order store

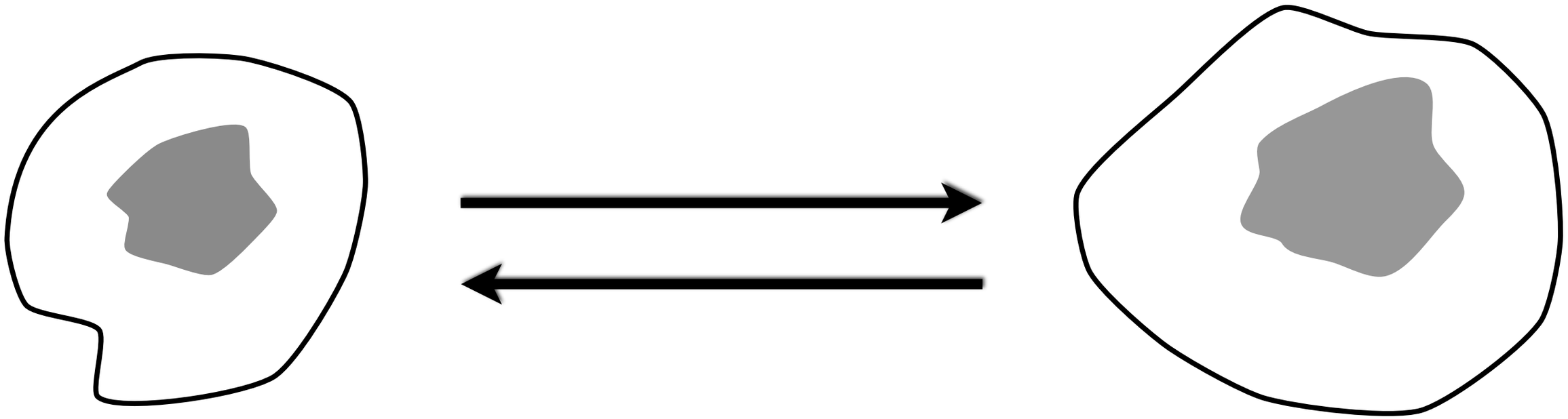
Strategies and games

- Game semantics models a *program* as a *strategy* for a *game*.
- Games define what moves are available.
- Constraints on strategies limit their behaviour.

Constraints and effects

Constraining strategies with:	means you don't get:
Innocence	Store
Bracketing	Control
Visibility	Higher-order store

Definability



Every program corresponds to a strategy

... and every (finite) strategy comes from a program

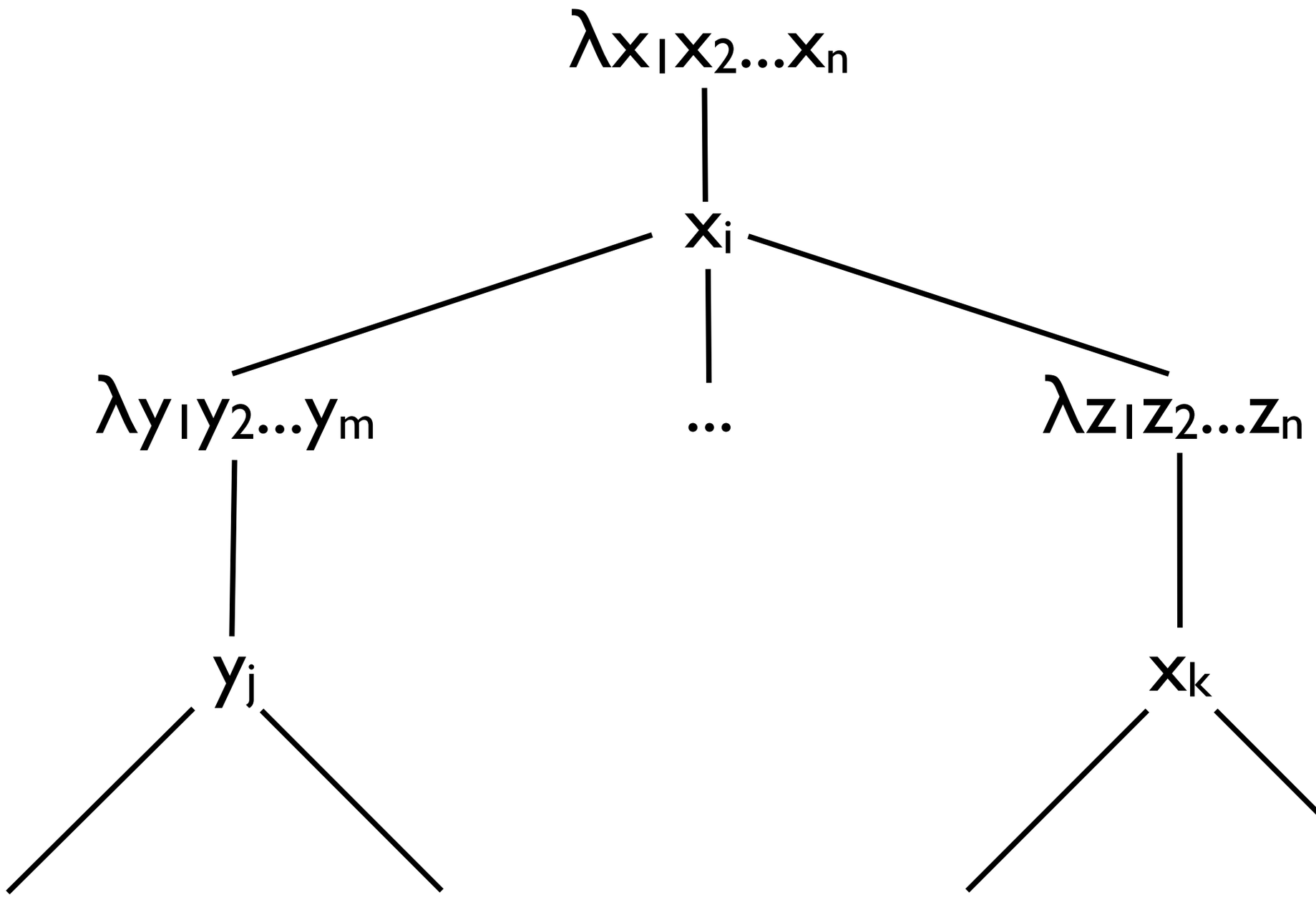
... and behavioural properties of strategies classify programs

1. Pure functional programs

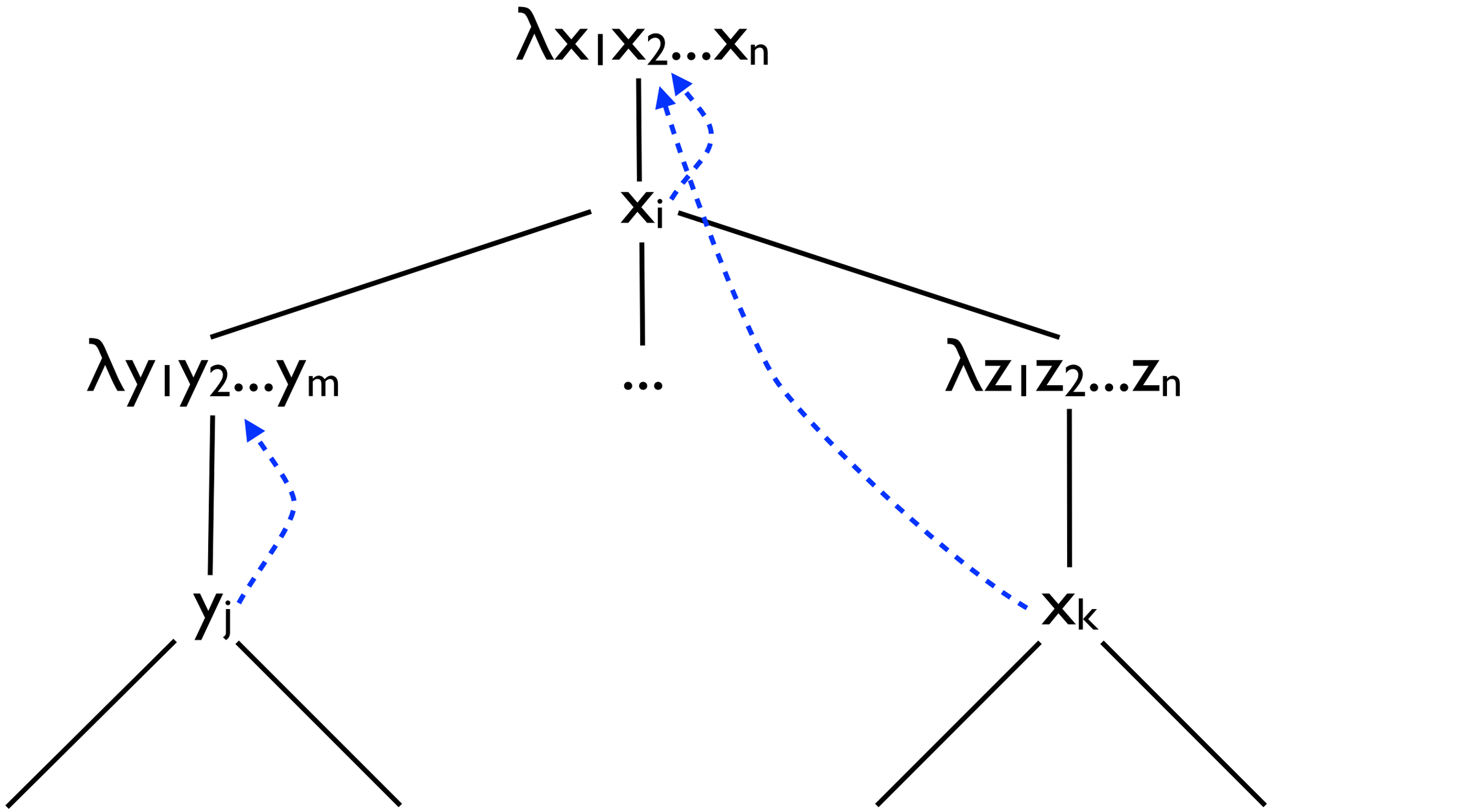
Typed lambda calculus

- Terms: $M ::= x \mid \lambda x.M \mid MM$
- Types: $A ::= \gamma \mid A \rightarrow A$
- Normal forms: $\lambda x_1 x_2 \ldots x_n \,.\, x_i \, M_1 M_2 \ldots M_k$

Normal forms as trees



Normal forms as trees

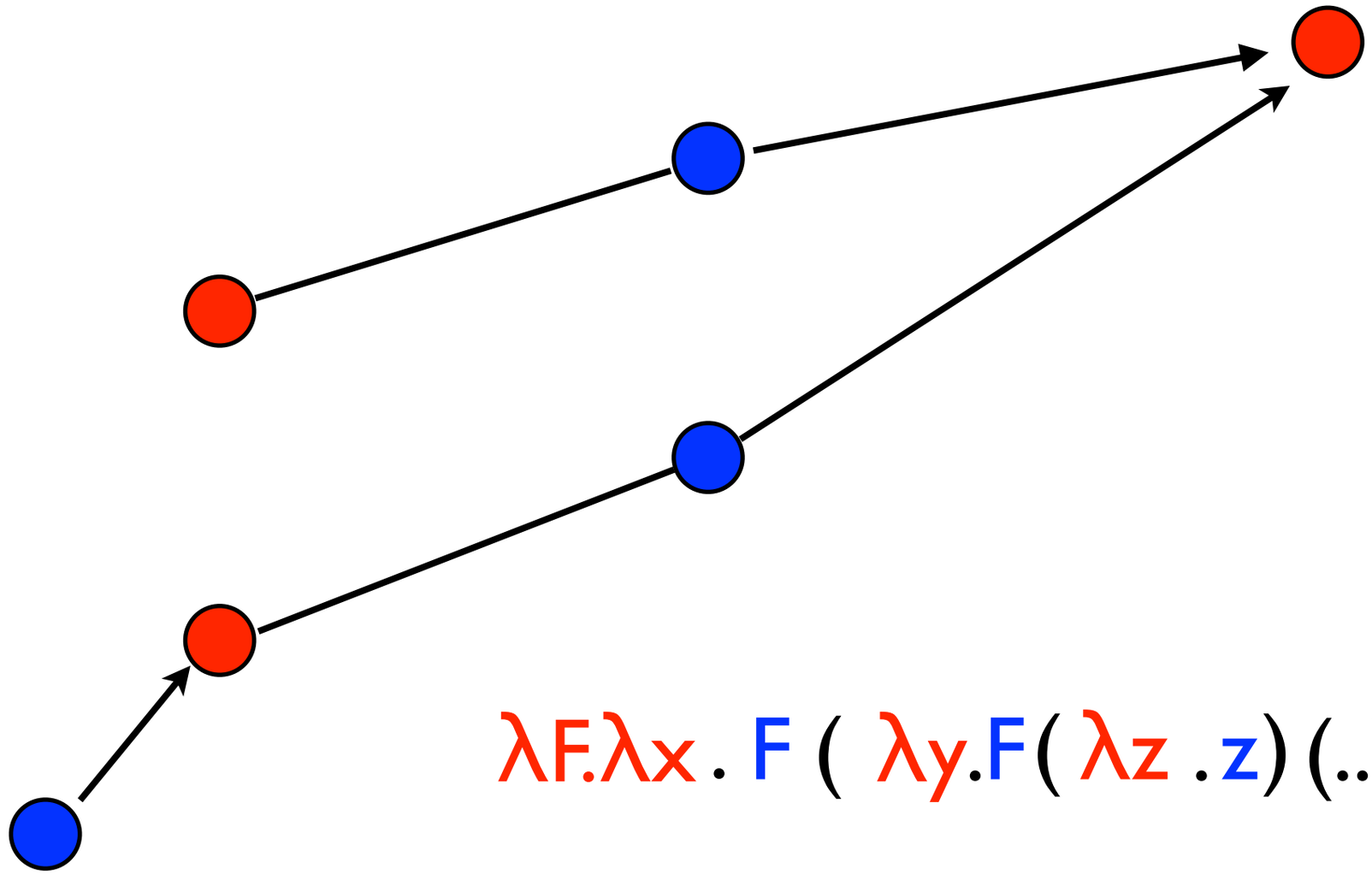


A path in the tree

- Root node
- Choice of variable
- Choice of argument
- Choice of variable
- ...

The path as a picture

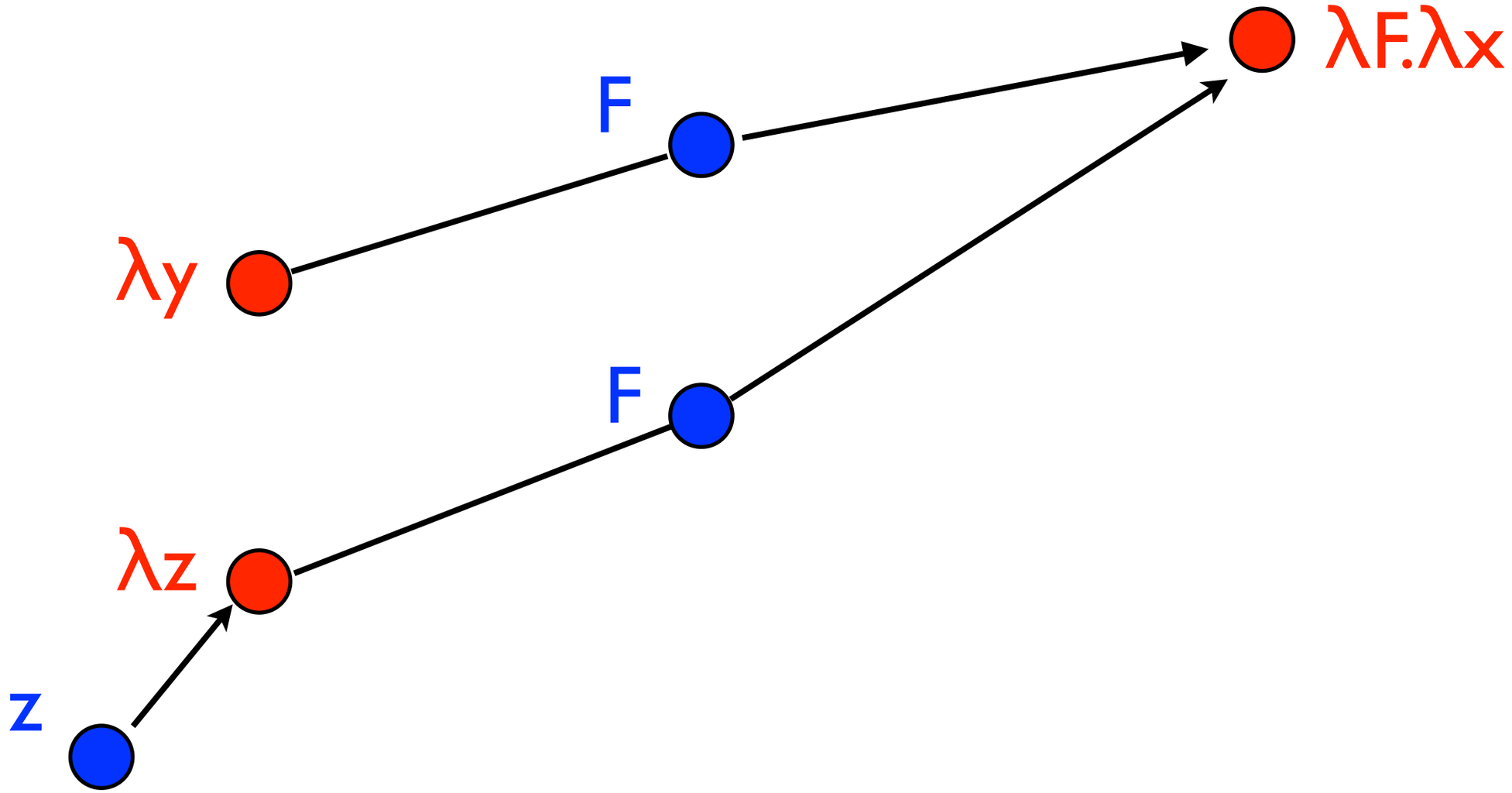
$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$



$$\lambda F.\lambda x\,.\,F\,(\,\lambda y.F(\,\lambda z\,.z)\,(..))\,(...)$$

The path as a picture

$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$



Games on arenas

- View the type as a tree: each → connects a node to its parent.
- Describe a term in normal form by playing a two-player game:
  - Opponent interrogates the term by choosing branches
  - Player represents the term by choosing head-variables

Games on arenas

Games on arenas

- At Opponent's turn, he chooses a descendent of Player's last move.

Games on arenas

- At Opponent's turn, he chooses a descendent of Player's last move.
- At Player's turn, he chooses a descendent of *any previous O-move*

Games on arenas

- At Opponent's turn, he chooses a descendent of Player's last move.
- At Player's turn, he chooses a descendent of *any previous O-move*
  - the *justification pointer* tells us which one

Strategies

- A *strategy* σ is a set of even-length plays of the game:
  - non-empty and even-prefix-closed
  - *deterministic:* sab, sac $\in \sigma \Rightarrow b=c$.
- Set of plays $\approx$ tree = (partial, infinite) normal form.

Definability...

- A strategy σ is *total* if it always has a response:

$$s \in \sigma, sa \text{ legal} \Rightarrow \exists\, sab \in \sigma$$

- Finite, total strategies correspond to normal forms.

... but not yet a model

- We can interpret every *normal form* as a strategy.
- We do not yet have an interpretation of arbitrary λ-terms.
- We want one! And we want it to be compositional.

Problem: asymmetry

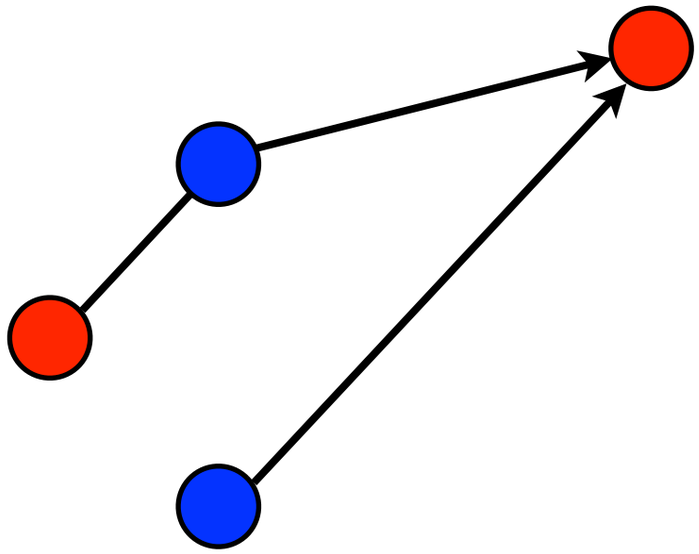
- These plays and strategies are *asymmetric*:
  - Opponent always has to move directly down the tree
    - Player can *backtrack*.
- Felscher (1985) referred to these as *E-strategies*.

Composition as interaction?

- The natural way to compose strategies would be by interaction.
- The asymmetry means our strategies cannot interact properly.

A failing interaction

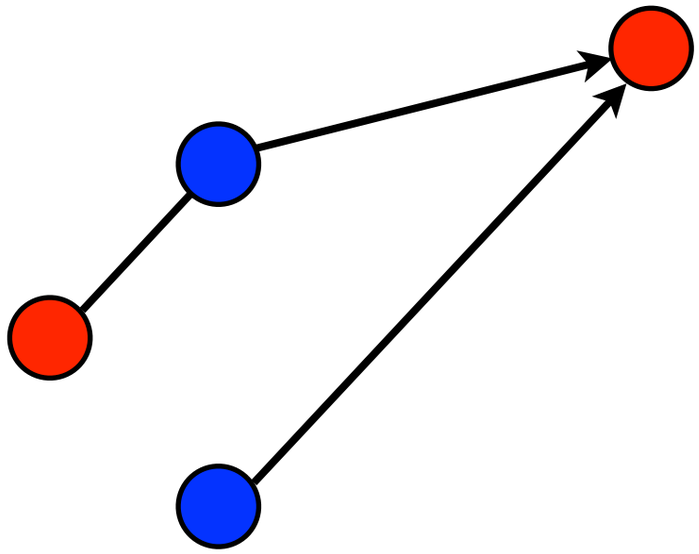
$((\gamma \to \gamma) \to (\gamma \to \gamma))$



$\lambda f.\lambda z.f(f(z))$

A failing interaction

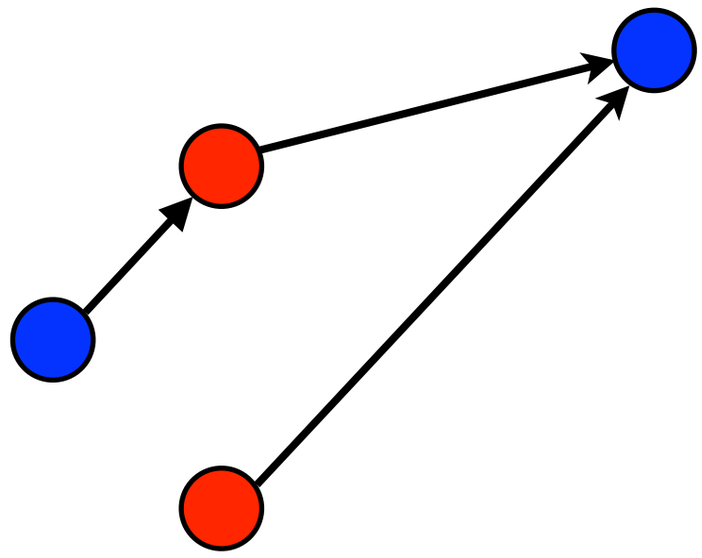
$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$



Supply $\lambda f.\lambda z.f(f(z))$ as argument to $\lambda F.\lambda x.F(\lambda y.y)(x)$

A failing interaction

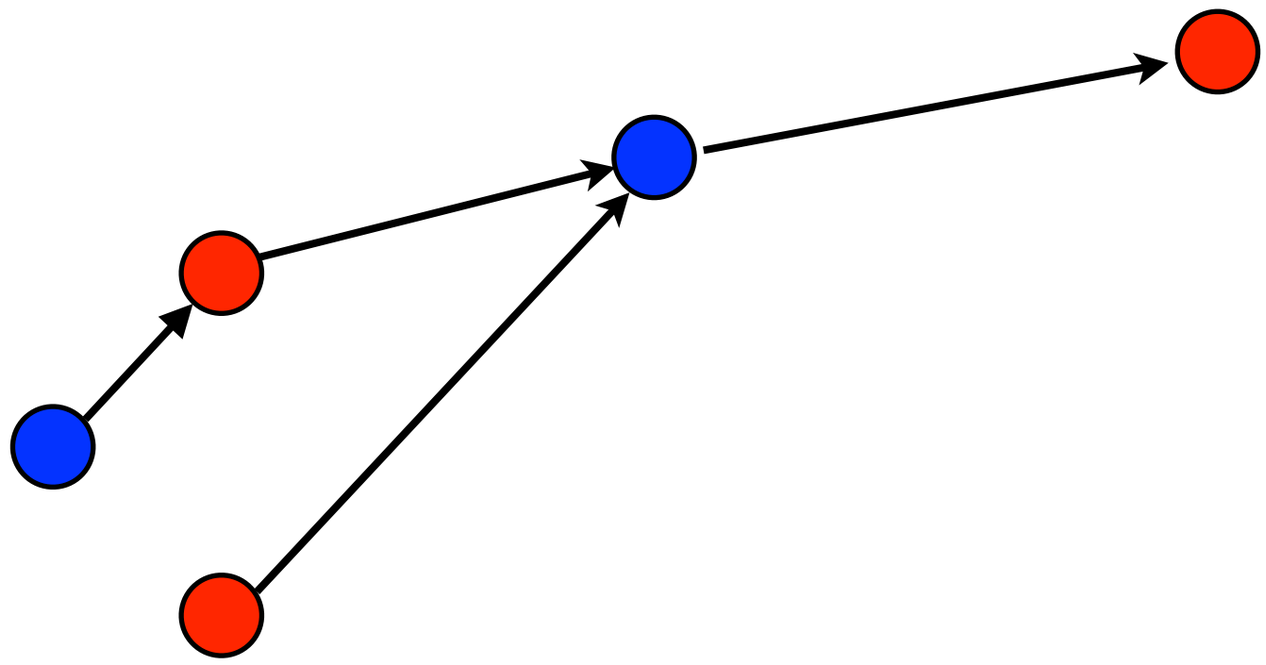
$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \quad \to \quad (\gamma \to \gamma)$$



Supply $\lambda f.\lambda z.f(f(z))$ as argument to $\lambda F.\lambda x.F(\lambda y.y)(x)$

A failing interaction

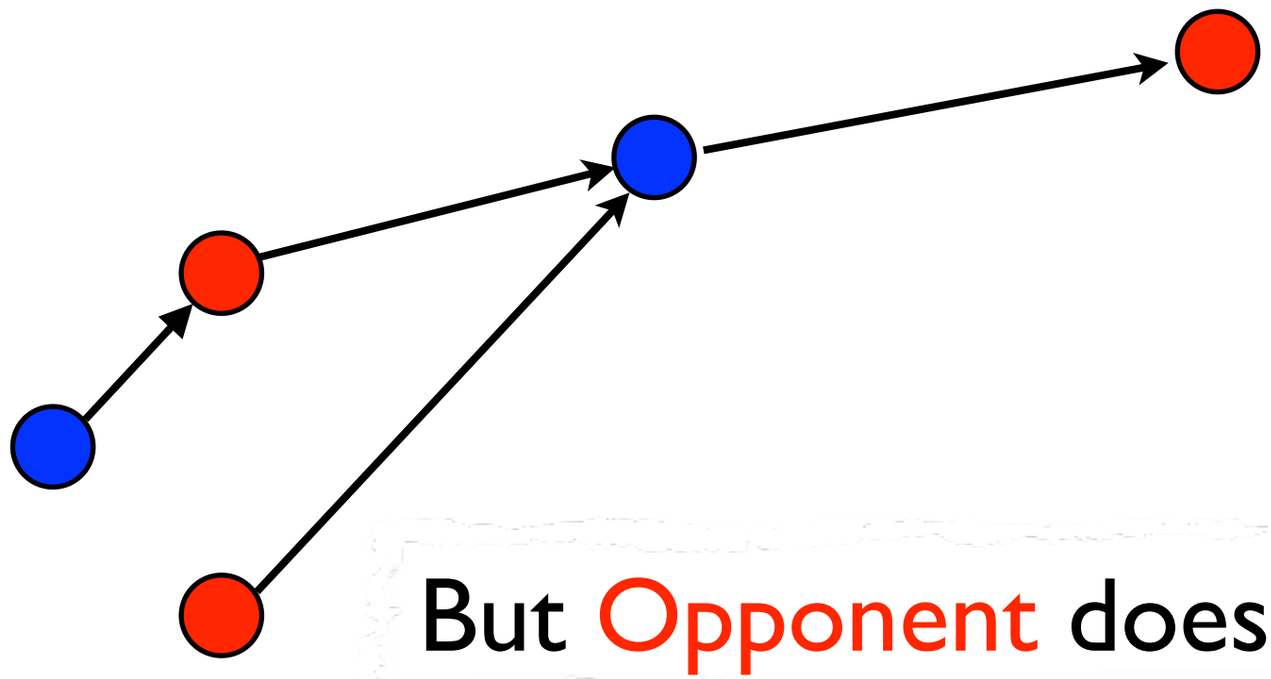
$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$



Supply $\lambda f.\lambda z.f(f(z))$ as argument to $\lambda F.\lambda x.F(\lambda y.y)(x)$

A failing interaction

$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$



But Opponent doesn't backtrack!

Supply $\lambda f.\lambda z.f(f(z))$ as argument to $\lambda F.\lambda x.F(\lambda y.y)(x)$

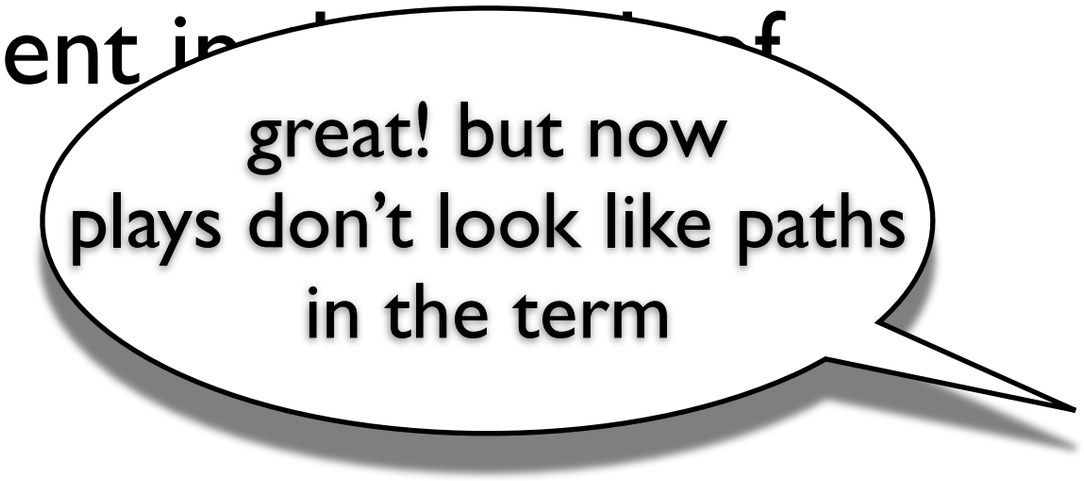
Restoring symmetry: innocence

Restoring symmetry: innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in the work of Coquand (JSL 1995).
- The idea is:
  - let both players backtrack

Restoring symmetry: innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in [illegible] of Coquand (JSL 1995).



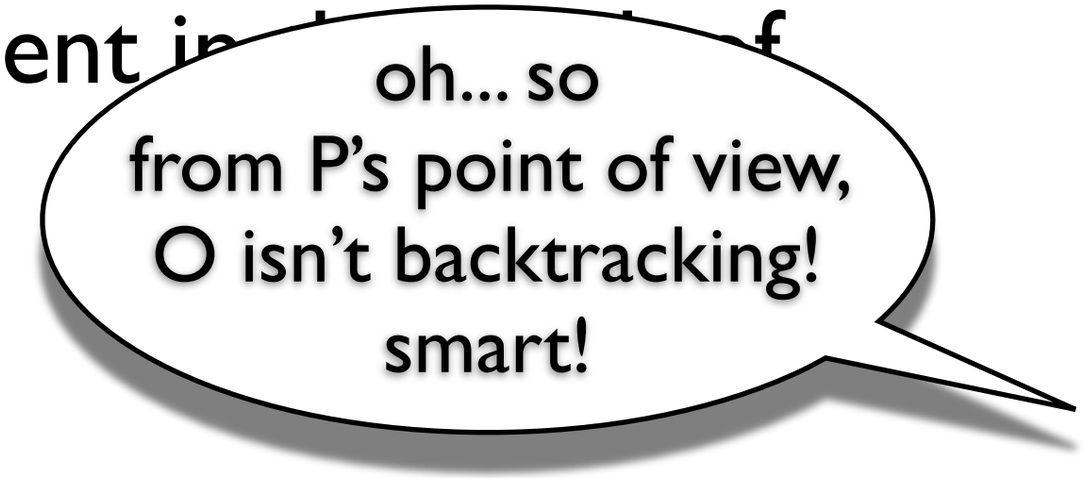
- The idea is:
  - let both players backtrack

Restoring symmetry: innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in the work of Coquand (JSL 1995).
- The idea is:
  - let both players backtrack
  - recover definability by constraining the

Restoring symmetry: innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in [illegible] of Coquand (JSL 1995).



- The idea is:
  - let both players backtrack
  - recover definability by constraining the

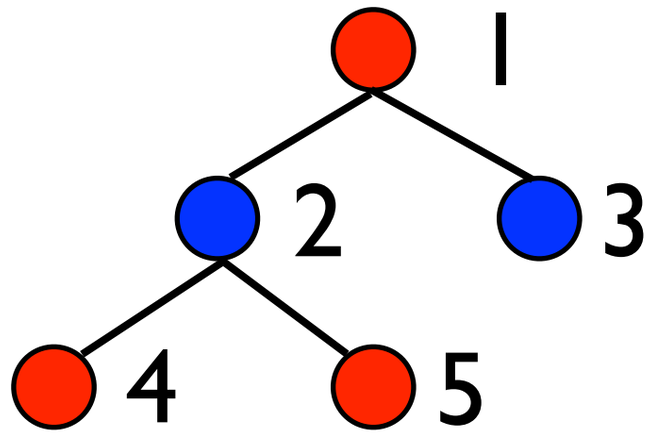
Arenas and plays

Arenas and plays

- An *arena* is a forest (collection of trees) of *moves*, labelled as Opponent and Player moves.
- O / P alternate down the trees.

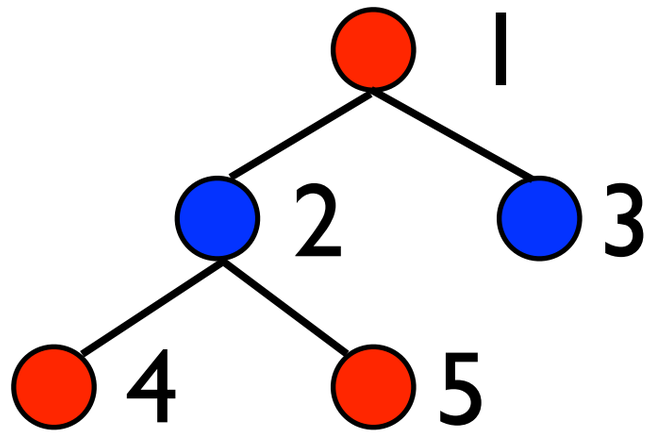
Arenas and plays

- An *arena* is a forest (collection of trees) of *moves*, labelled as Opponent and Player moves.
- O / P alternate down the trees.



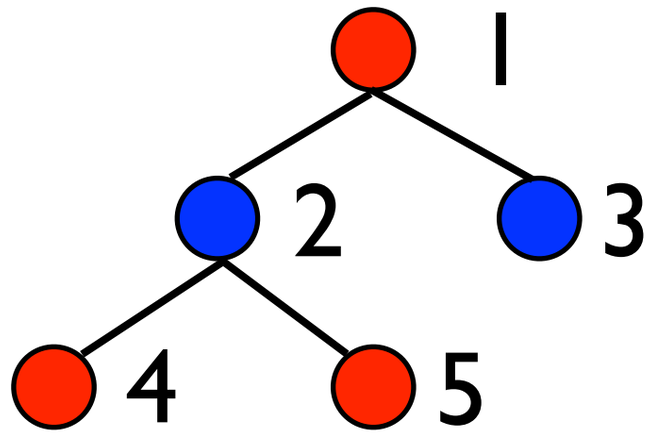
Arenas and plays

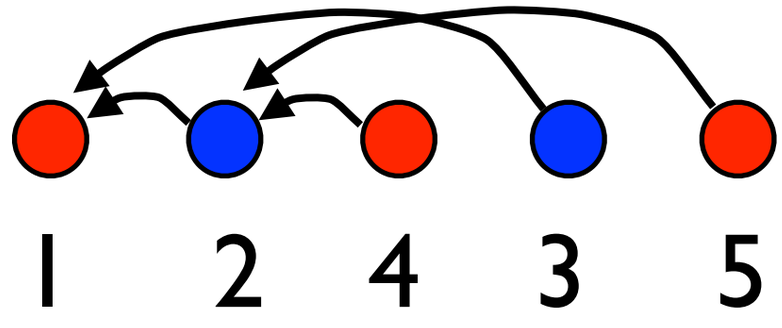
- An *arena* is a forest (collection of trees) of *moves*, labelled as Opponent and Player moves.
- O / P alternate down the trees.
- A *play* is a sequence of moves-with-pointers.
- Pointer-chains are paths in the arena.



Arenas and plays

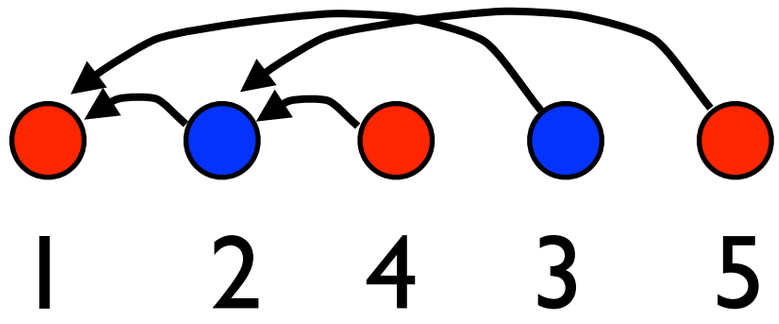
- An *arena* is a forest (collection of trees) of *moves*, labelled as Opponent and Player moves.
- O / P alternate down the trees.
- A *play* is a sequence of moves-with-pointers.
- Pointer-chains are paths in the arena.





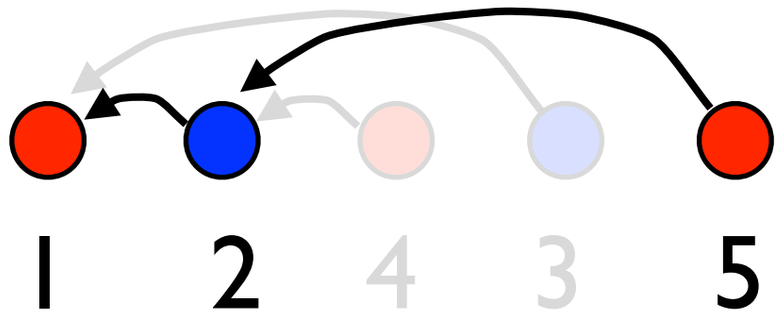
Views

- We want Player to behave as though O were not backtracking.
- At any point in the play, we can *erase* certain moves to give a *P-view* which disguises backtracking:



Views

- We want Player to behave as though O were not backtracking.
- At any point in the play, we can *erase* certain moves to give a *P-view* which disguises backtracking:



Innocent strategy

- An *innocent strategy* σ on an arena is a set of even-length plays such that:
  - σ is non-empty and closed under even-prefix
  - σ is deterministic
  - if $sab \in \sigma$, $t \in \sigma$, and view(sa) = view (ta) then $tab \in \sigma$

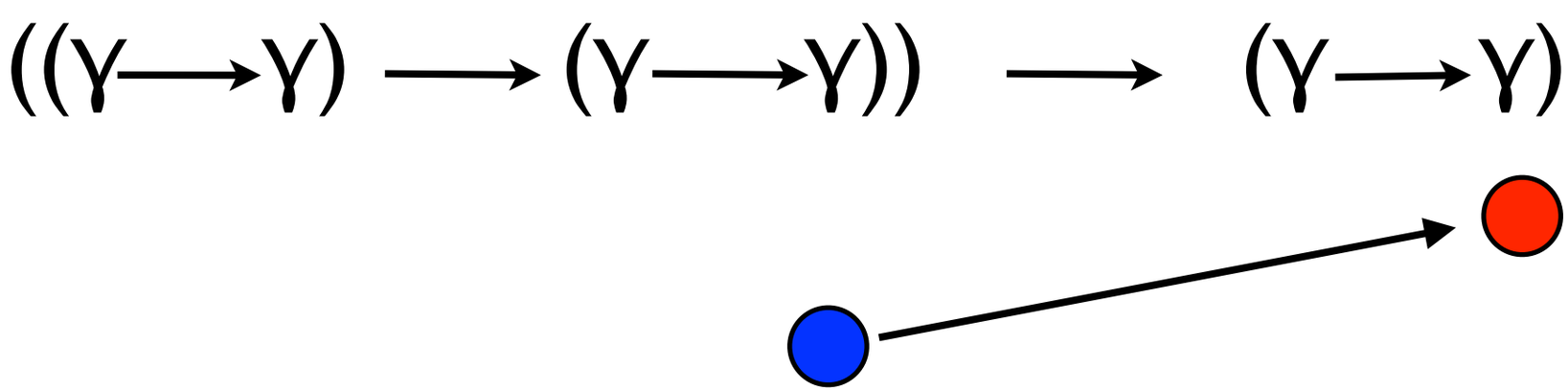
Example: a non-innocent strategy

$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$

Example: a non-innocent strategy

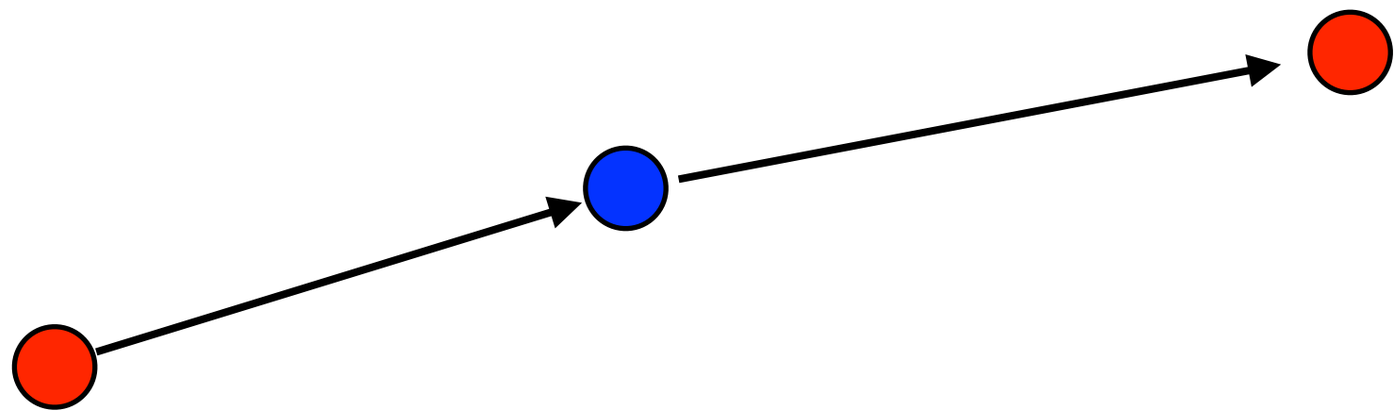
$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$

Example: a non-innocent strategy



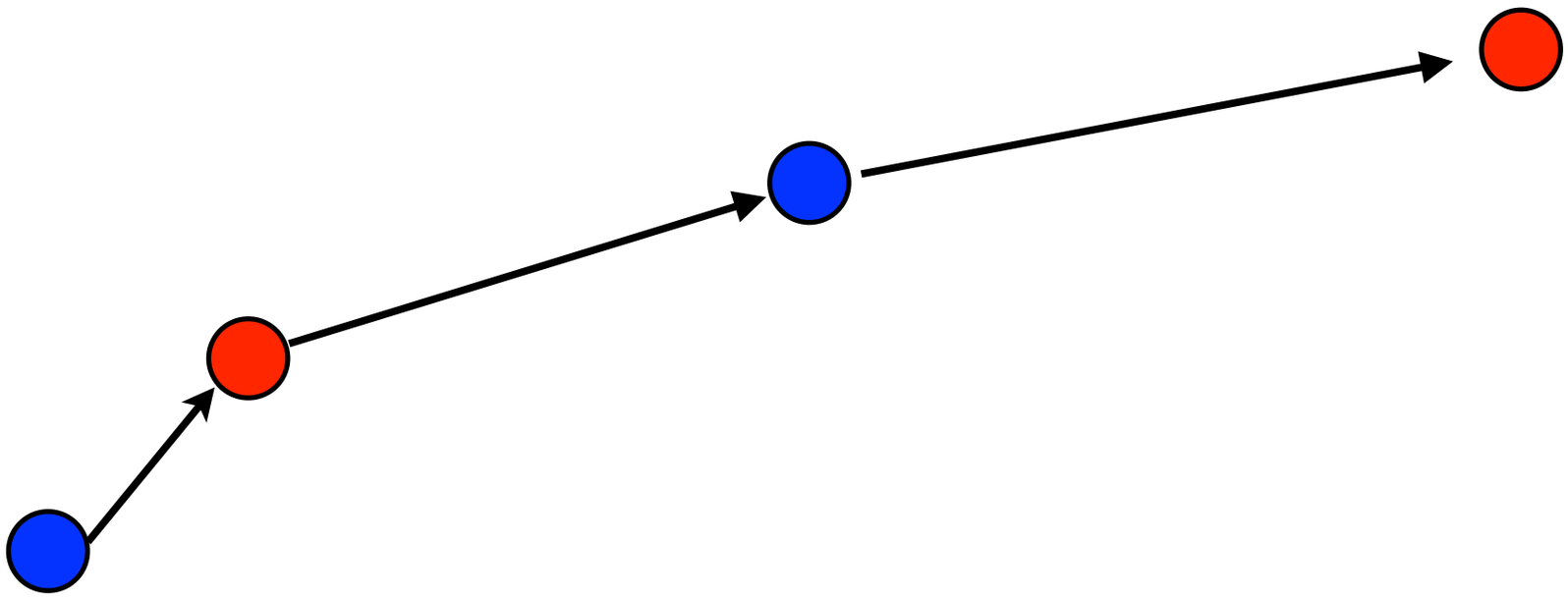
Example: a non-innocent strategy

$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$



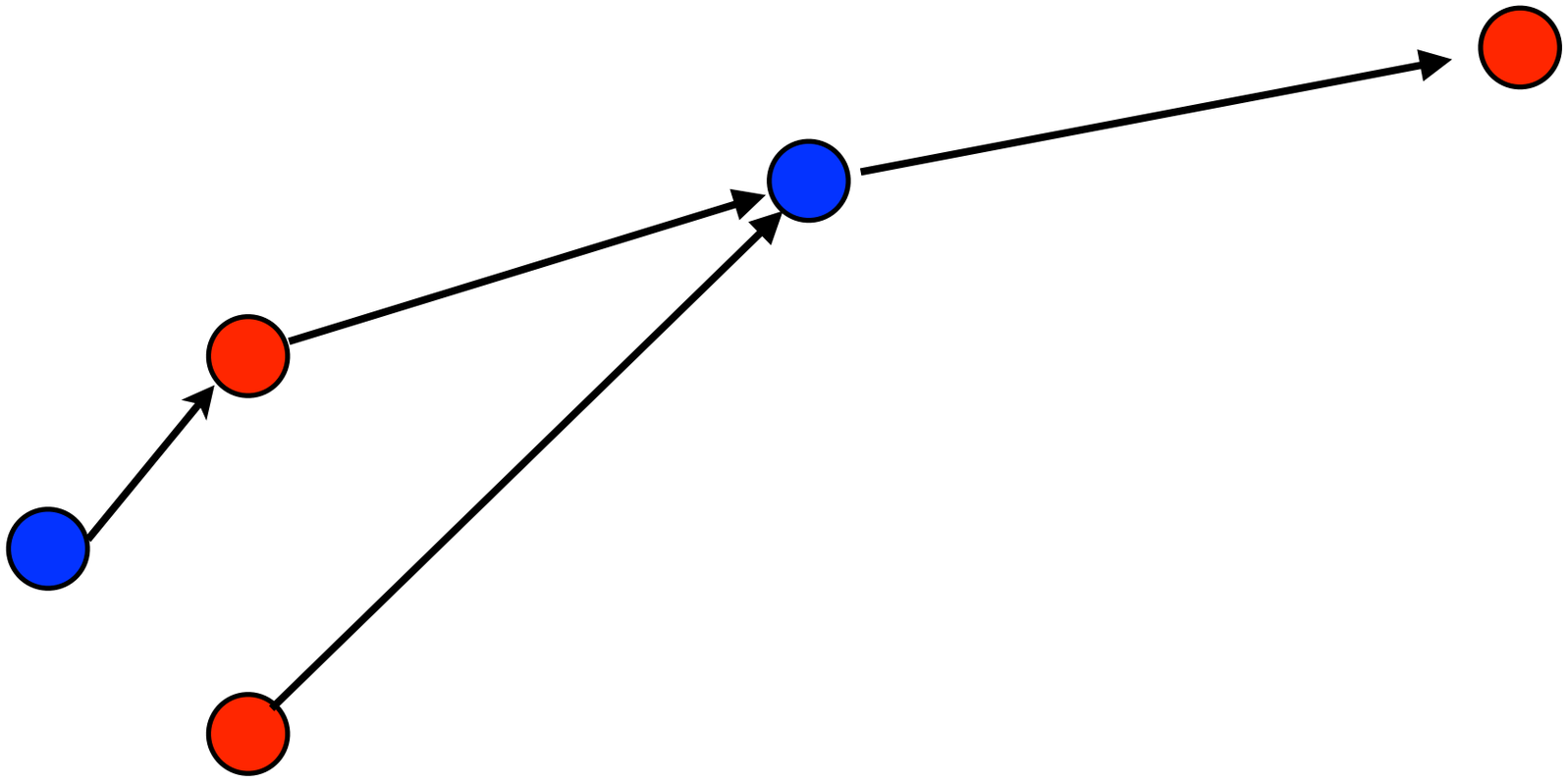
Example: a non-innocent strategy

$$((\gamma \longrightarrow \gamma) \longrightarrow (\gamma \longrightarrow \gamma)) \longrightarrow (\gamma \longrightarrow \gamma)$$



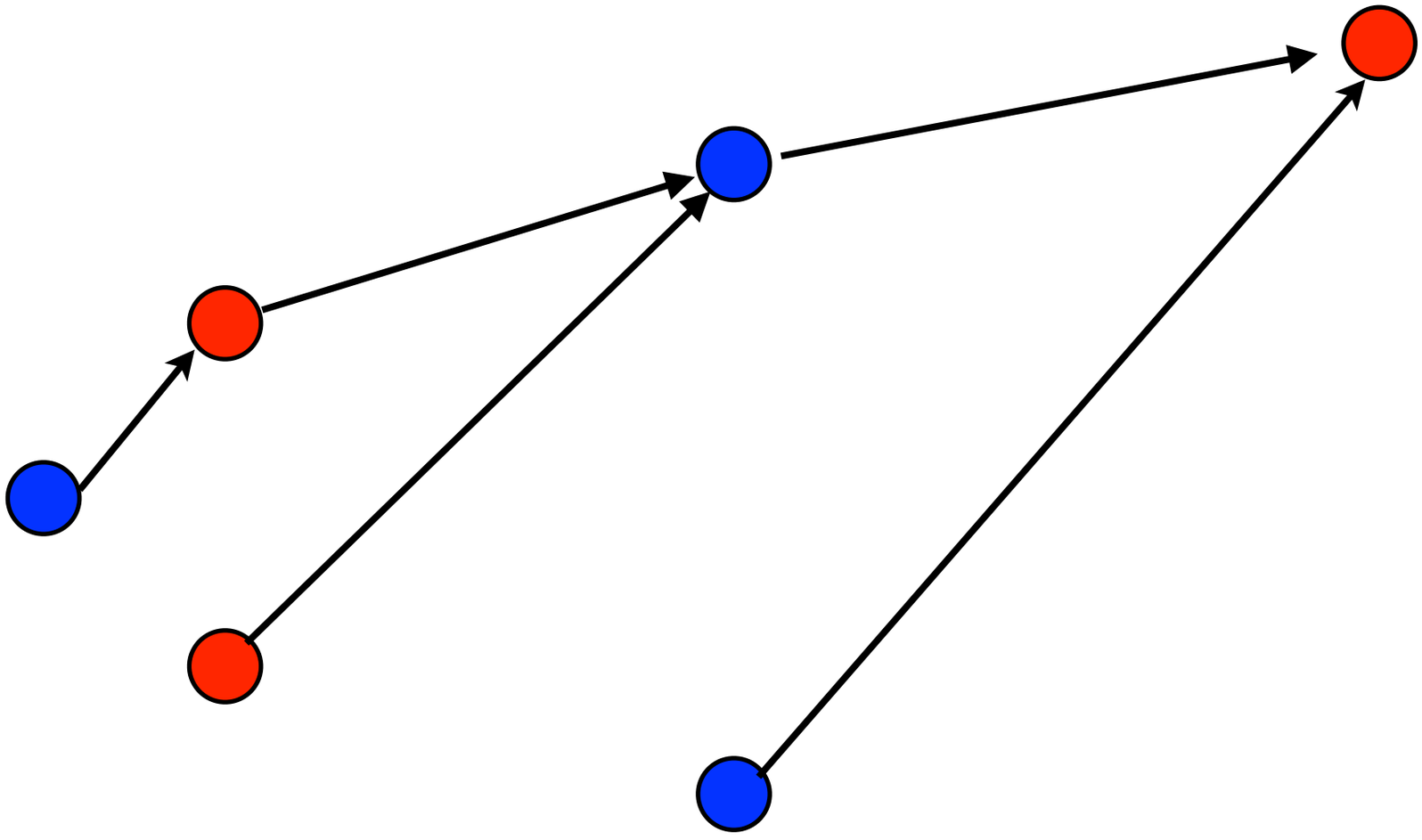
Example: a non-innocent strategy

$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$



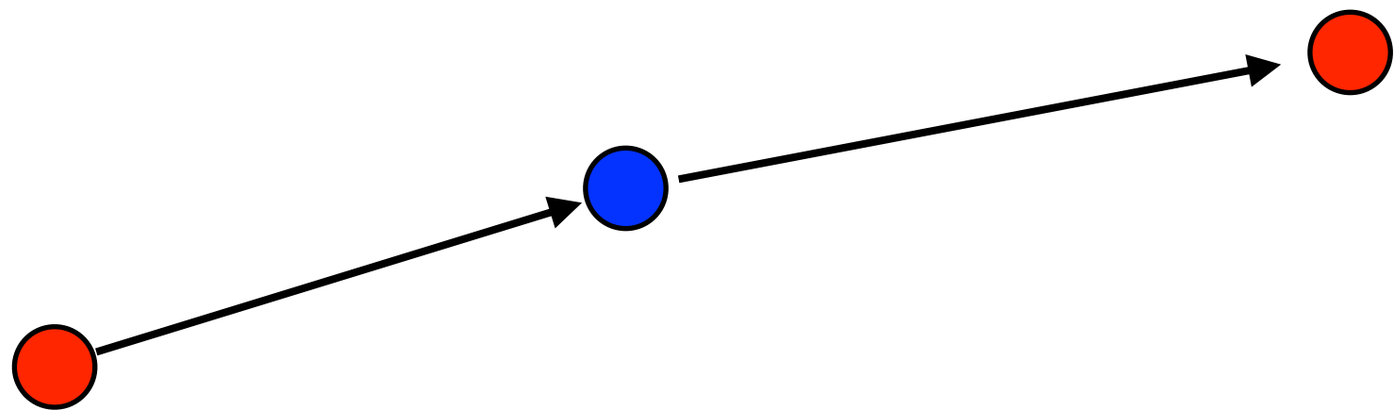
Example: a non-innocent strategy

$((Y \to Y) \to (Y \to Y)) \to (Y \to Y)$



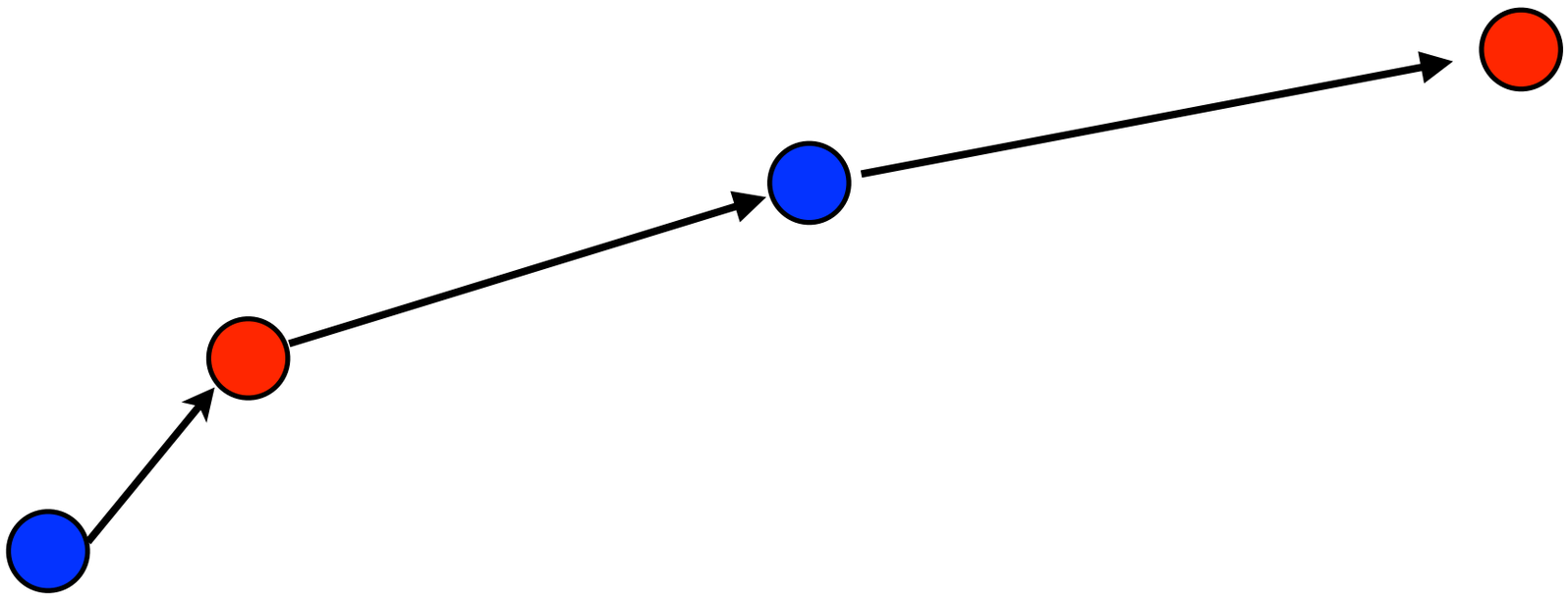
Example: a non-innocent strategy

$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$



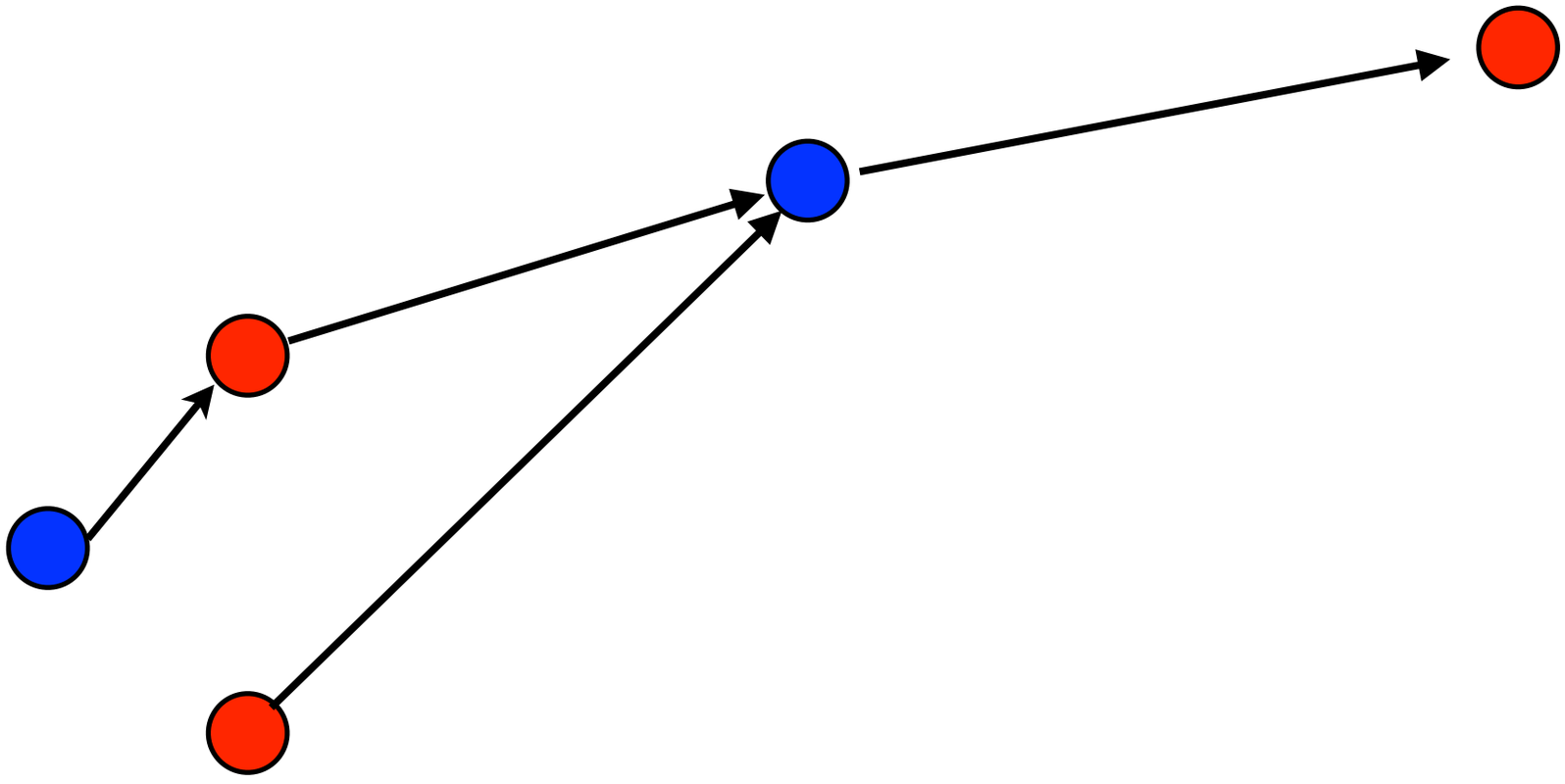
Example: a non-innocent strategy

$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$



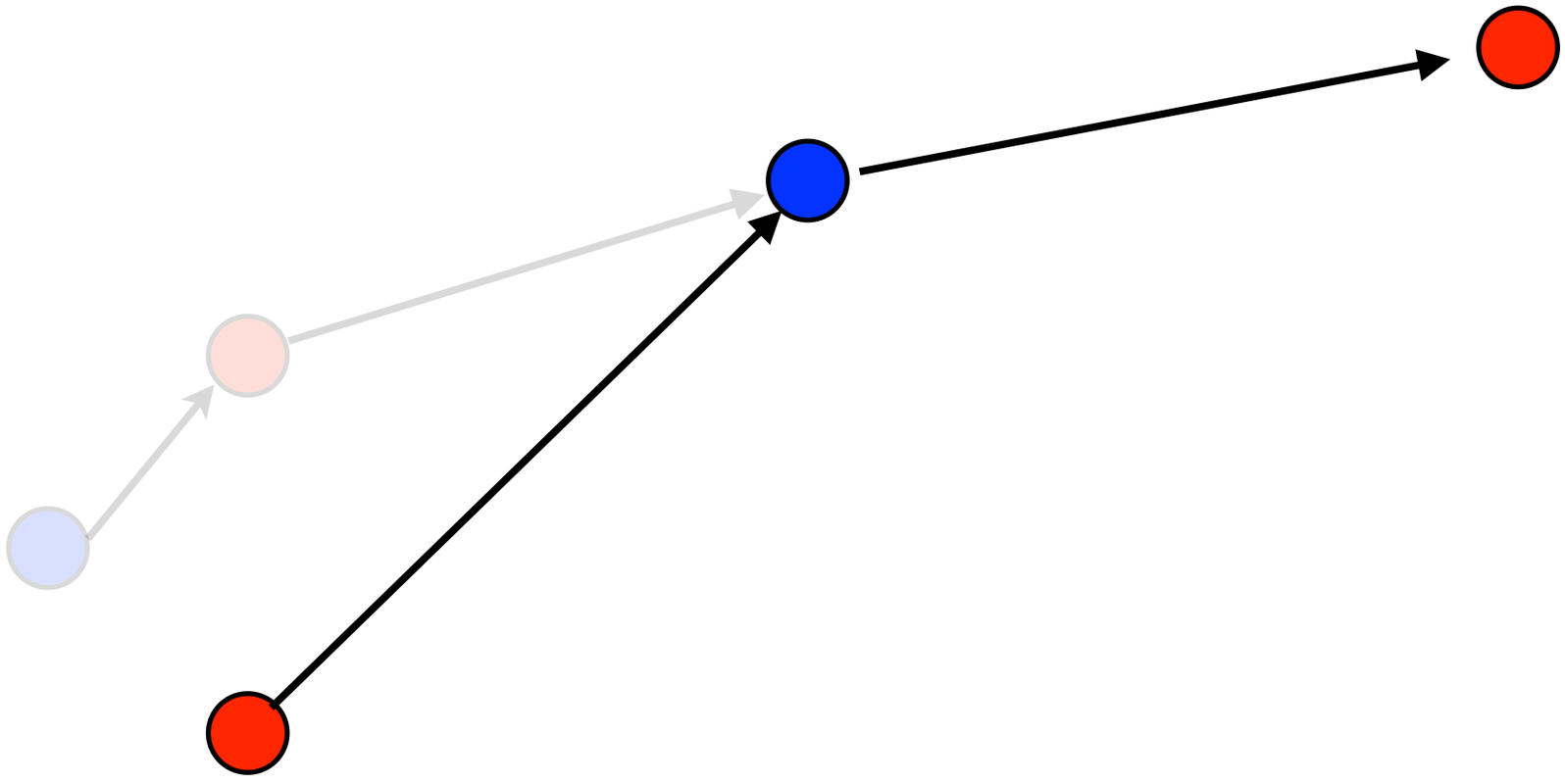
Example: a non-innocent strategy

$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$



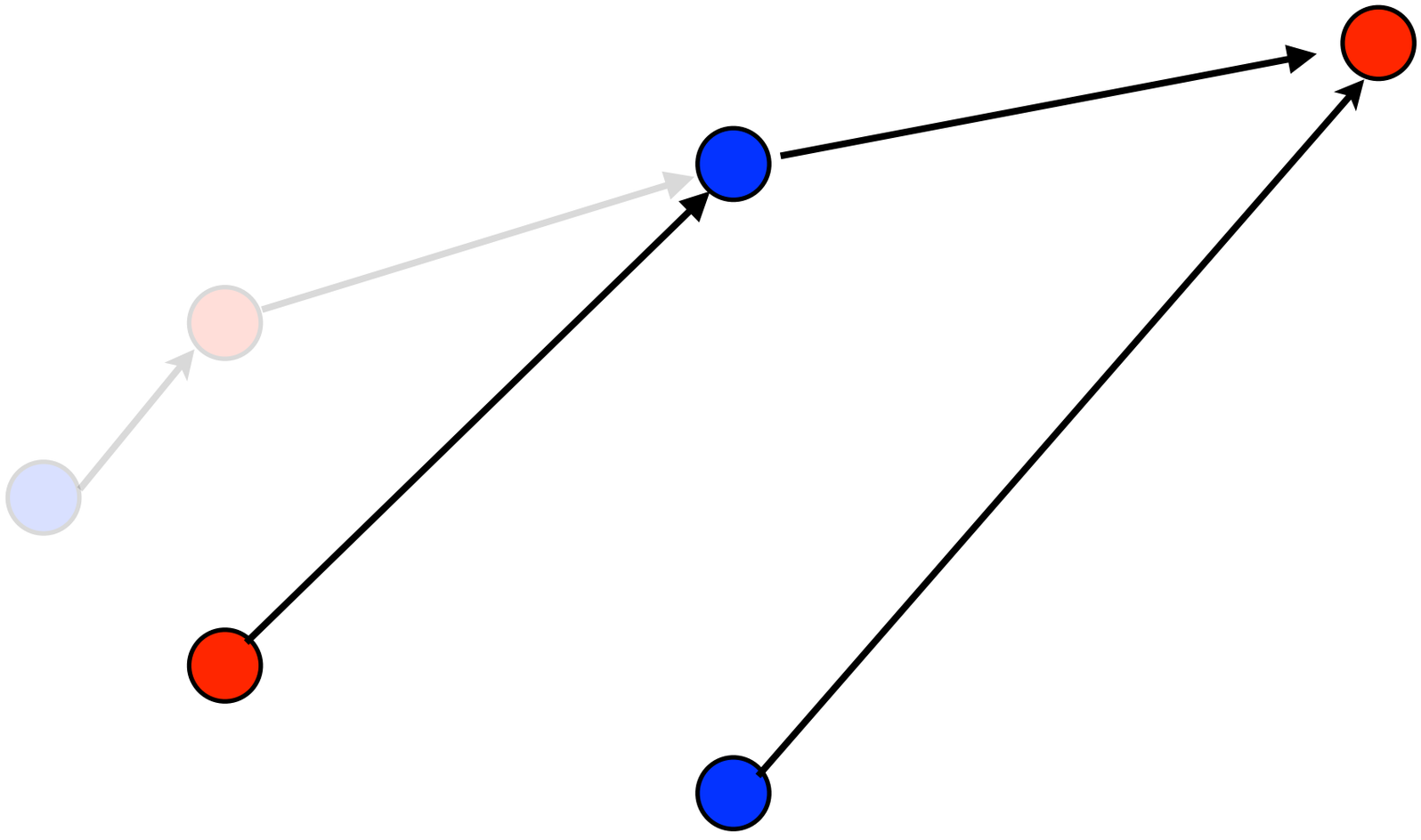
Example: a non-innocent strategy

$$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$$



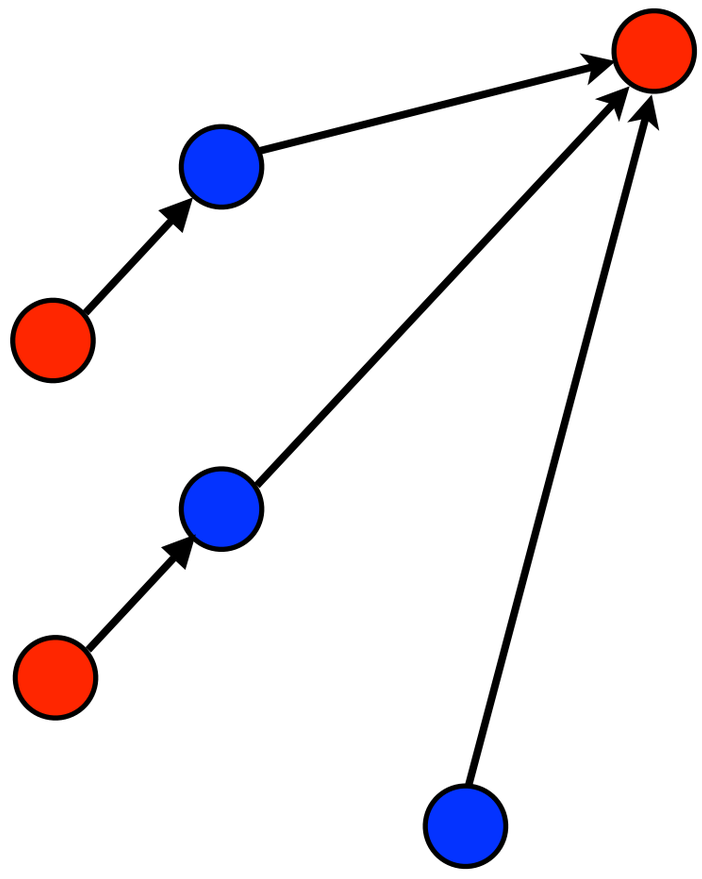
Example: a non-innocent strategy

$((Y \longrightarrow Y) \longrightarrow (Y \longrightarrow Y)) \longrightarrow (Y \longrightarrow Y)$



We have a model

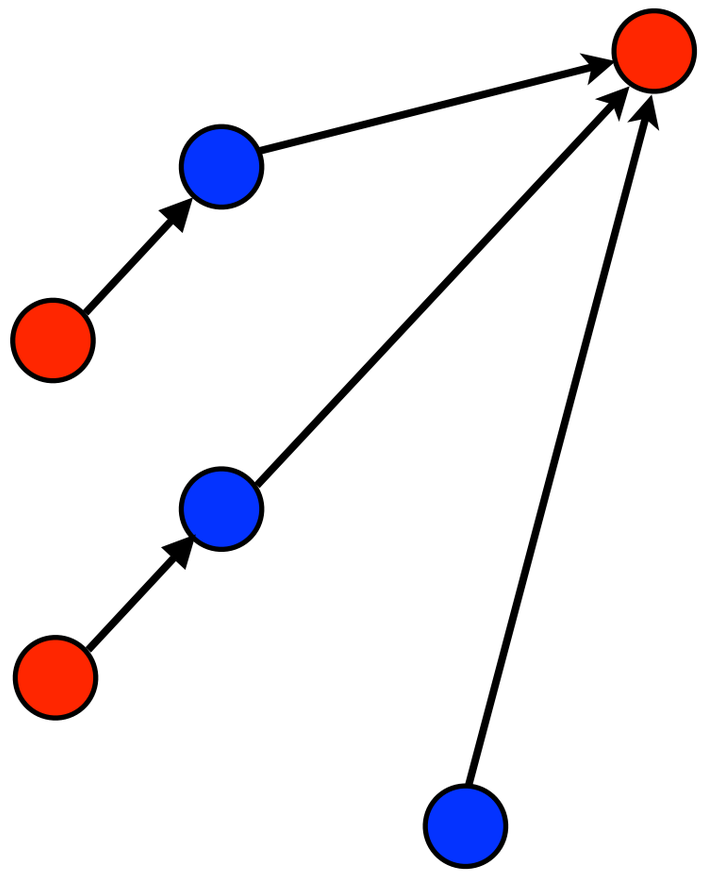
((γ→γ) →(γ→γ))



(λf.λz.f(f(z)))

We have a model

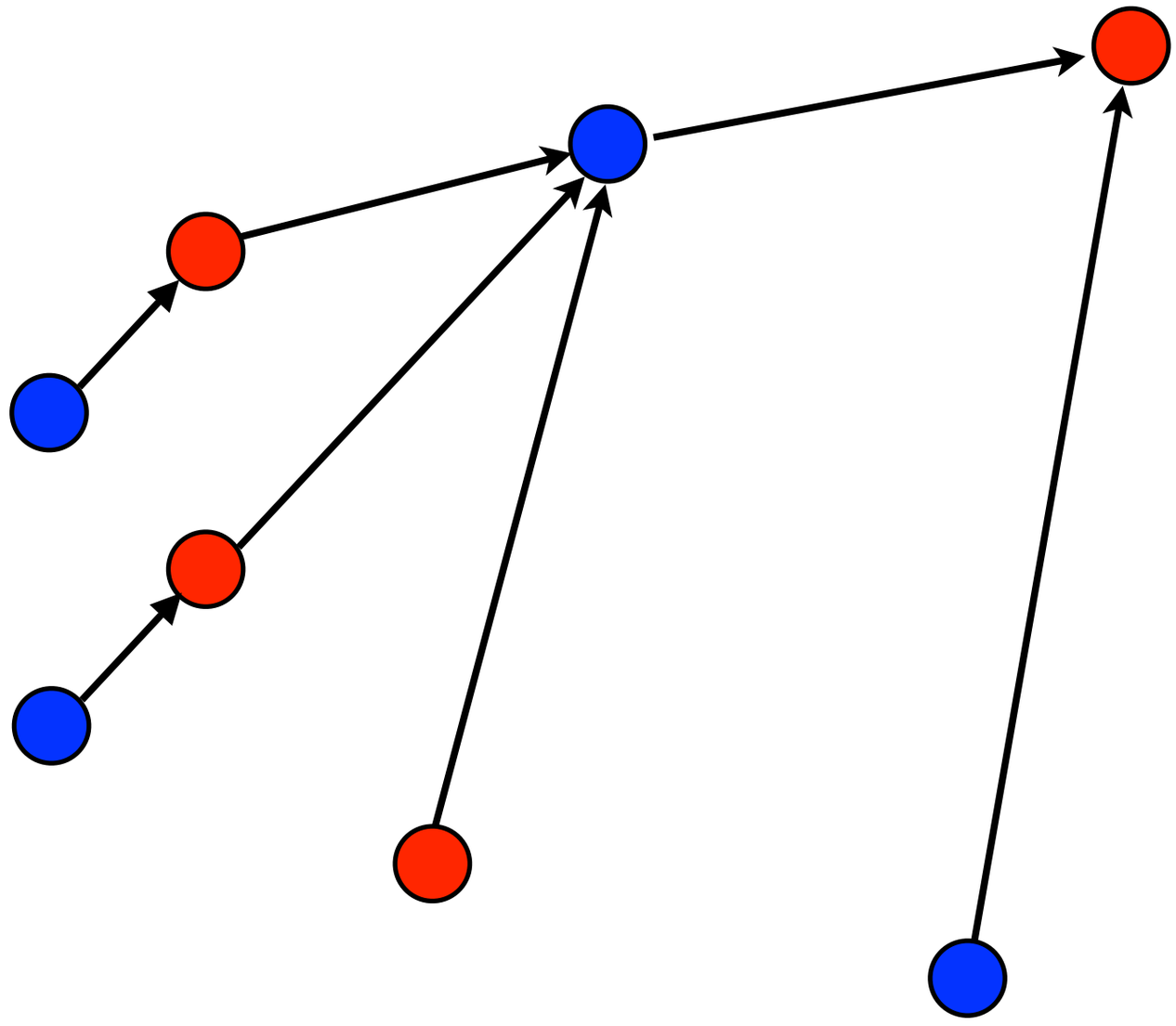
((γ→γ) →(γ→γ))



(λf.λz.f(f(z)))

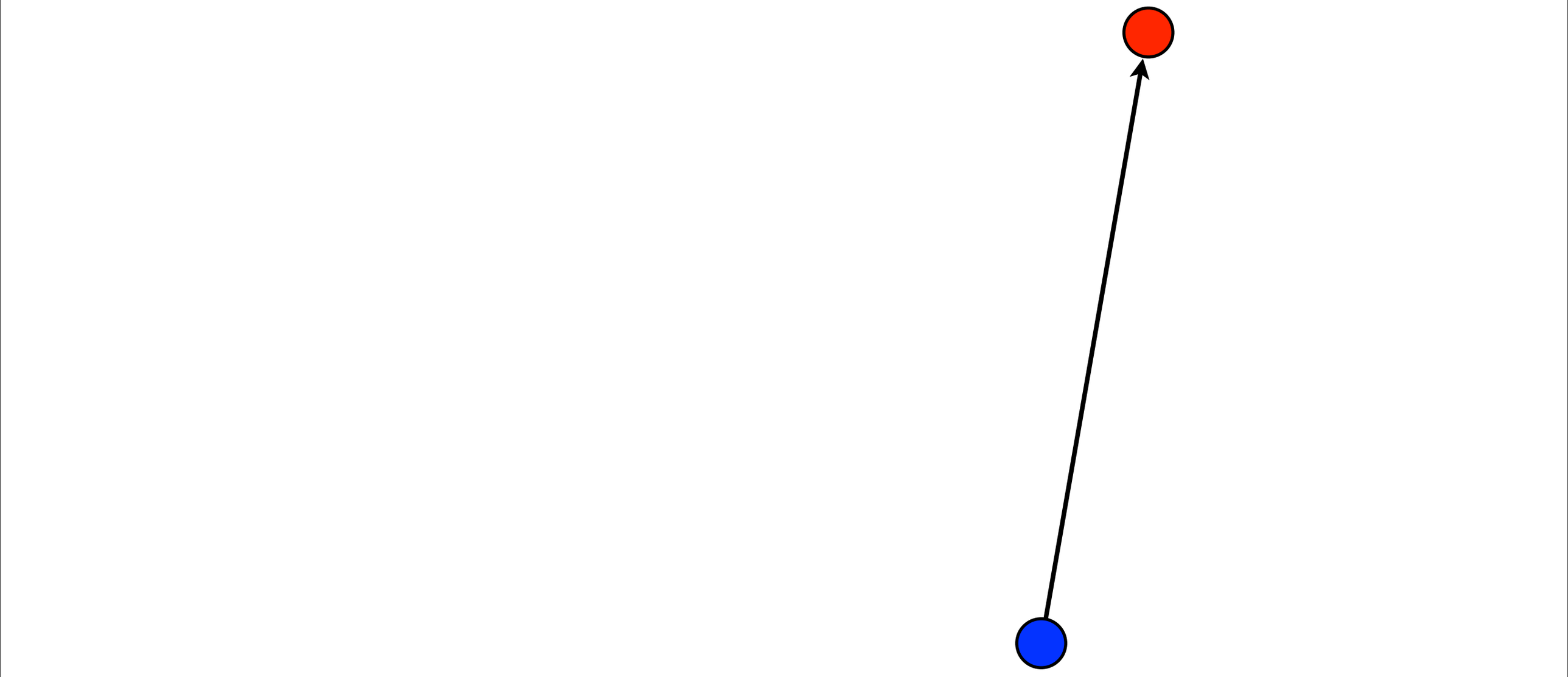
We have a model

$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$



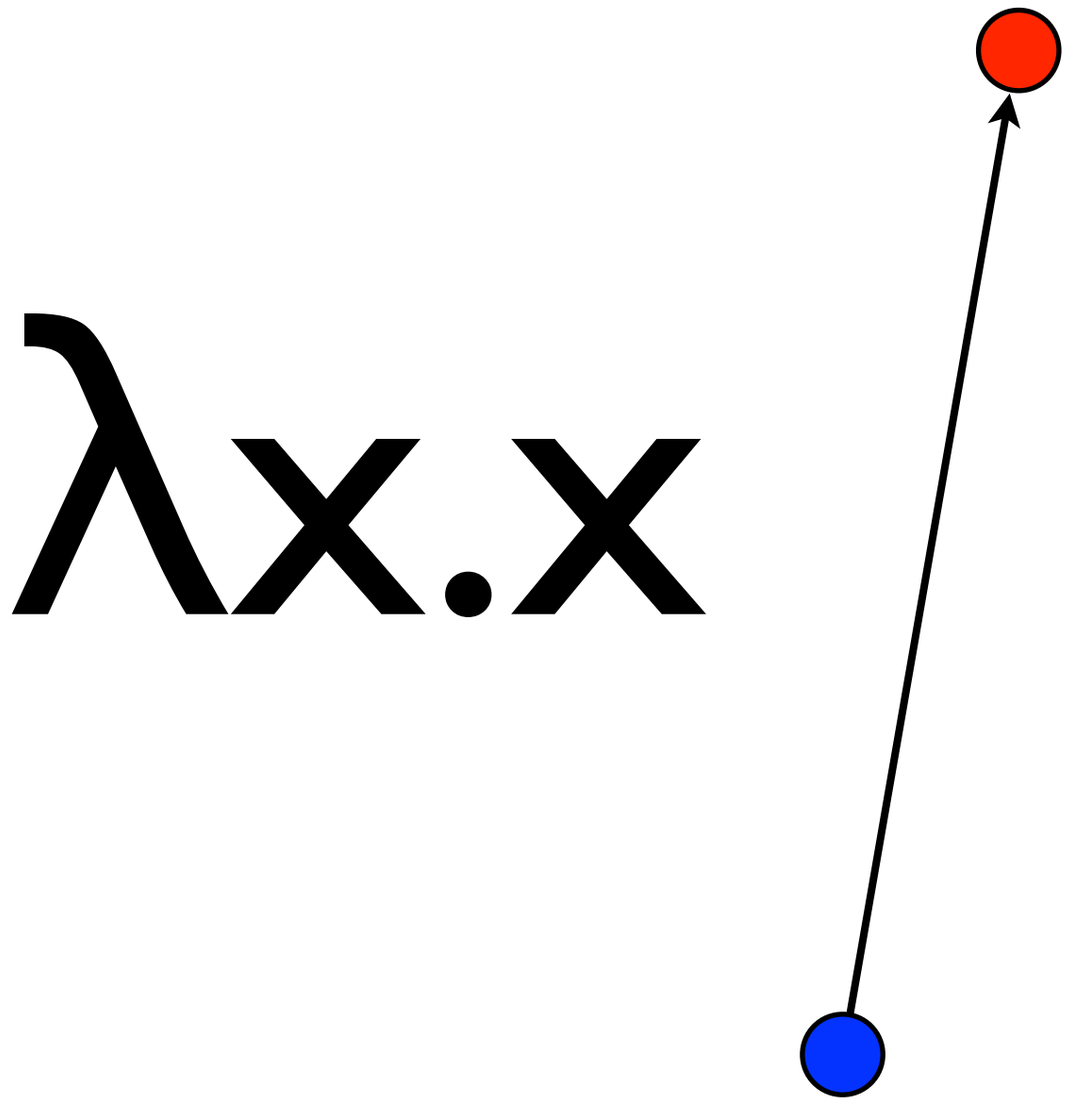
$(\lambda F.\lambda x.F(\lambda y.y)(x))$ $(\lambda f.\lambda z.f(f(z)))$

We have a model



We have a model

$(\gamma \to \gamma)$



$(\lambda F.\lambda x.F(\lambda y.y)(x))$ $(\lambda f.\lambda z.f(f(z)))$

Soundness

Fact [cf. Hyland-Ong 2000]

If $M : A \to B$ and $N : A$ are normal forms, the strategy obtained by

- allowing the strategies for M and N to interact
- hiding the play in A

is the total, innocent strategy corresponding to the normal form of MN.

Soundness

- Proving soundness is hard work.
- We approach it by showing that innocent strategies have the structure of a Cartesian closed category.
- CCCs are just what is needed to make a sound model of the λ-calculus.

A category

- We can build a category of arenas and innocent strategies:
  - Objects are arenas
  - Morphisms are innocent strategies
  - Composition is interaction plus hiding
  - Identity is the *copycat strategy*

A category

- We can build a category of arenas and innocent strategies:
  - Objects are arena
  - Morphisms are inn
  - Composition is interaction plus hiding
  - Identity is the *copycat strategy*

you
have to show that
this preserves
innocence!

A category

- We can build a category of arenas and innocent strategies:
  - Objects are arenas
  - Morphisms are innocent strategies
  - Composition is interaction plus hiding
  - Identity is the *copycat strategy*

Copycat strategies

Copycat strategies

$\lambda f.f$

$((\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma))$

Copycat strategies

$\lambda f.f$

$((\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma))$

Copycat strategies

$\lambda f.\, \lambda x.fx$

$((\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma))$

Copycat strategies

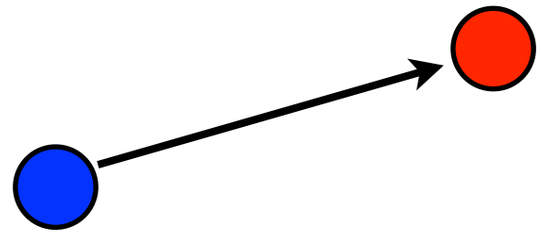
$\lambda f.\, \lambda x.fx$

$((\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma))$

Copycat strategies

$\lambda f.\, \lambda x.fx$

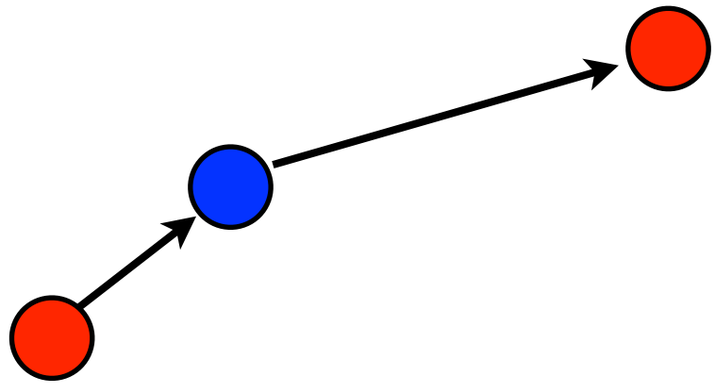
$((\gamma \to \gamma) \to (\gamma \to \gamma))$



Copycat strategies

$\lambda f.\, \lambda x.fx$

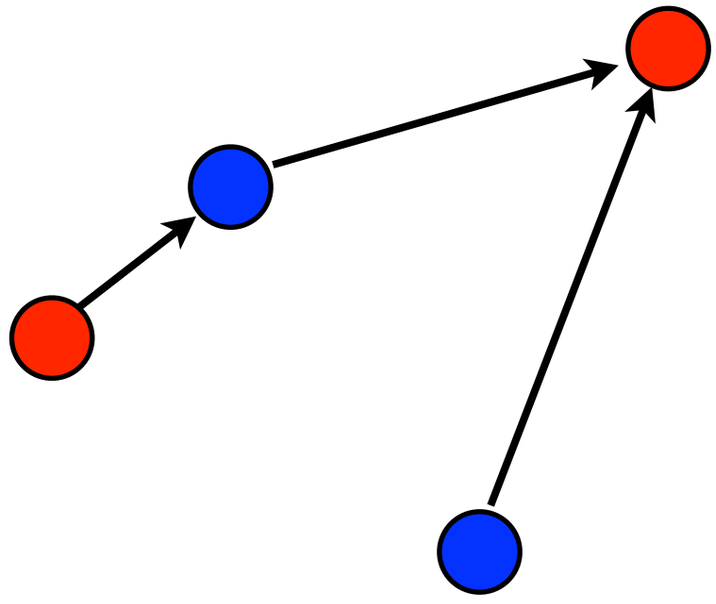
$((\gamma \to \gamma) \to (\gamma \to \gamma))$



Copycat strategies

$\lambda f.\, \lambda x.fx$

$((\gamma \to \gamma) \to (\gamma \to \gamma))$



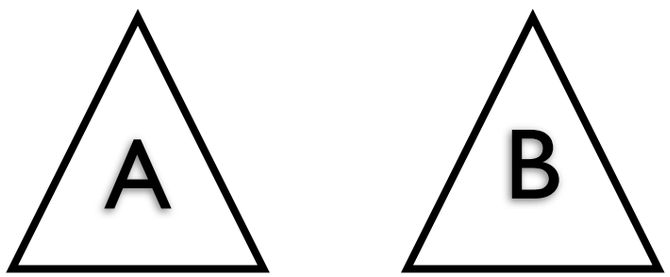
Cartesian closure

Finally we show that the category has products and exponentials:

Cartesian closure

Finally we show that the category has products and exponentials:

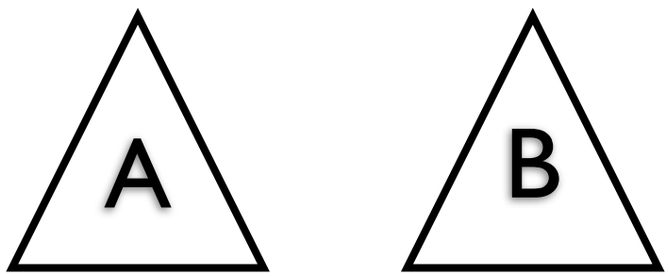
$$A \times B$$



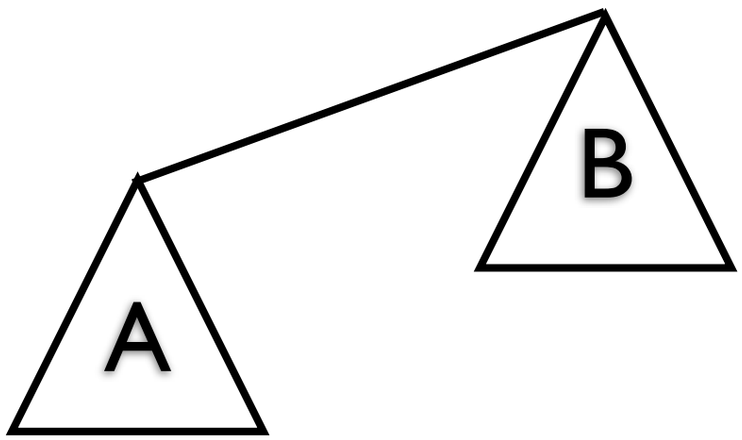
Cartesian closure

Finally we show that the category has products and exponentials:

$A \times B$



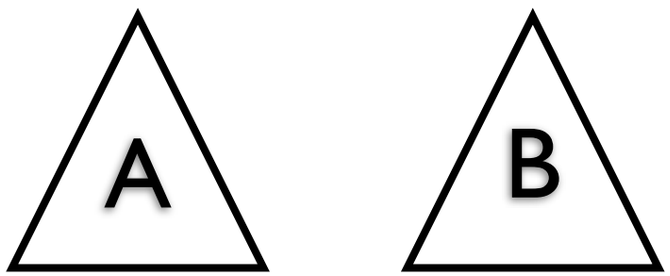
$A \Rightarrow B$



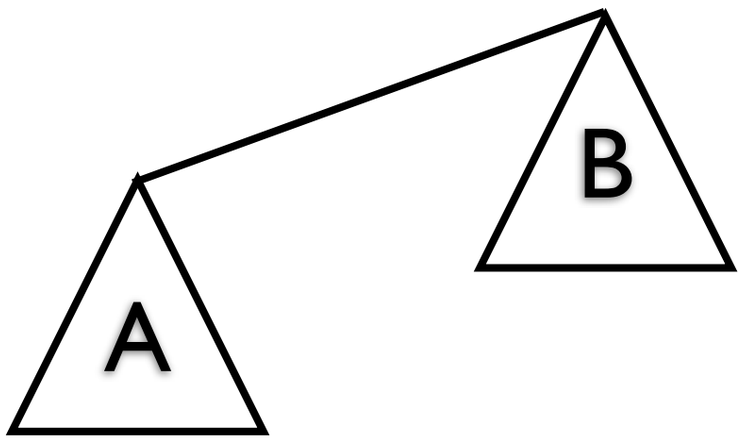
Cartesian closure

Finally we show that the category has products and exponentials:

$A \times B$



$A \Rightarrow B$



After proving that these behave properly, we get soundness.

Full completeness

Full completeness

- *Full completeness* (definability) of the model is now *easy:*

Full completeness

- *Full completeness* (definability) of the model is now *easy:*
  - an innocent strategy is determined by the E-strategy it contains

Full completeness

- *Full completeness* (definability) of the model is now e*asy:*
  - an innocent strategy is determined by the E-strategy it contains
  - finite, total E-strategies correspond to terms.

Literature note 1: abstract machines

The soundness of interaction of strategies as a model of λ-application means that it can be used as a kind of *abstract machine* for computation. See e.g.

Danos, Herbelin and Regnier, *Games Semantics and Abstract Machines, 1996.*

Curien and Herbelin, *Computing with Abstract Böhm Trees, 1998.*

Literature note 2: innocence semantically

- We have presented innocence as a syntactically-inspired condition.
- The work of Melliès on *Asynchronous Games* shows that it can be recovered from semantically-inspired considerations to do with *permutability* of moves.

Melliès, *Asynchronous Games 2: The True Concurrency of Innocence*, 2006.

2. Adding data and recursion: PCF

PCF

- So far we have only considered *logic* rather than programming.
- Plotkin's language PCF is a prototypical functional programming language.
  - Typed λ-calculus with base types for numeric and boolean values.
  - Constants for arithmetic and boolean operations.
  - Recursion.

PCF

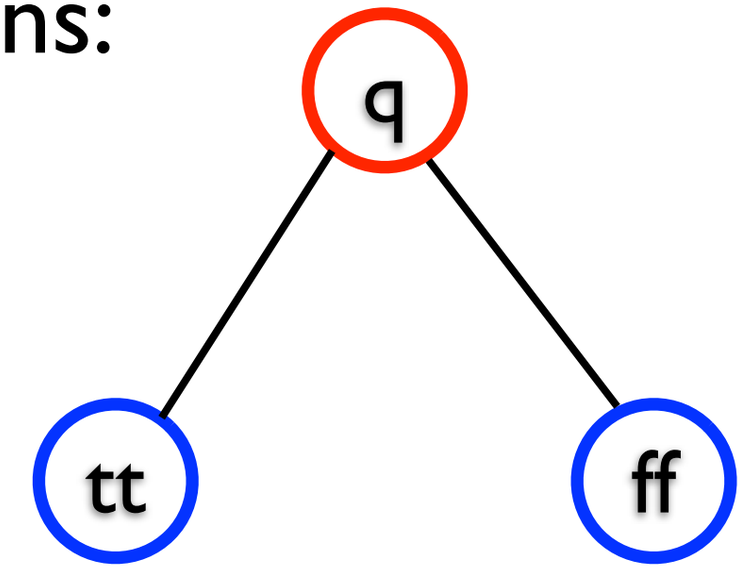
- Types: A ::= **N** | **B** | A → A
- Terms: M ::= x | λx.M | MM

  | n | succ | pred | true | false ...

  | if-then-else | Y
- We will focus on PCF over the Booleans.

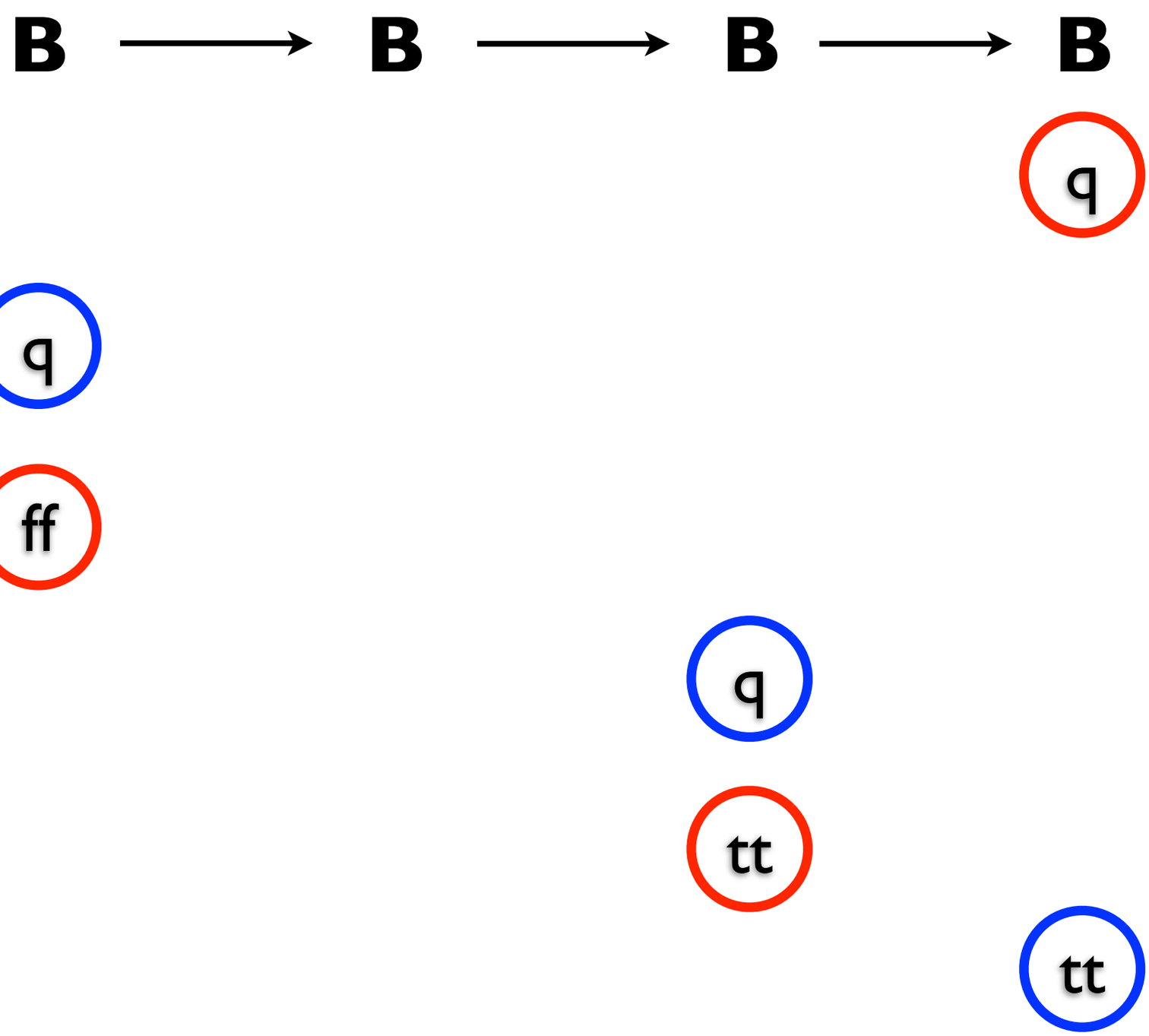
A Game for the Booleans

- We introduce an arena with *data* to model the Booleans:



- Note: this is the same as the arena for the type $\gamma \to \gamma \to \gamma$ which encodes Booleans in the λ-calculus.

if-then-else



Exercise

- In the λ-calculus, write down the term corresponding to if-then-else, when Booleans are encoded using $\gamma \to \gamma \to \gamma$
  - true is $\lambda x.\lambda y.x$, false is $\lambda x.\lambda y.y$
  - if-then-else (over Booleans) is...?
- Compare the corresponding strategy to the one just indicated.

Recursion

- Strategies are sets of plays.
- Directed unions of innocent strategies are innocent strategies, and composition preserves them.
- We can therefore define *least fixed points*, which lets us interpret recursion in the usual way.
- Of course, we abandon *totality*.

Definability for PCF?

- The CCC of arenas and innocent strategies therefore contains a model of PCF.
- Does it have the definability property?
- Is there a class of "normal form" PCF terms that correspond to the finite innocent strategies?

Normal Forms

- What might a normal form look like?
- Something like this:

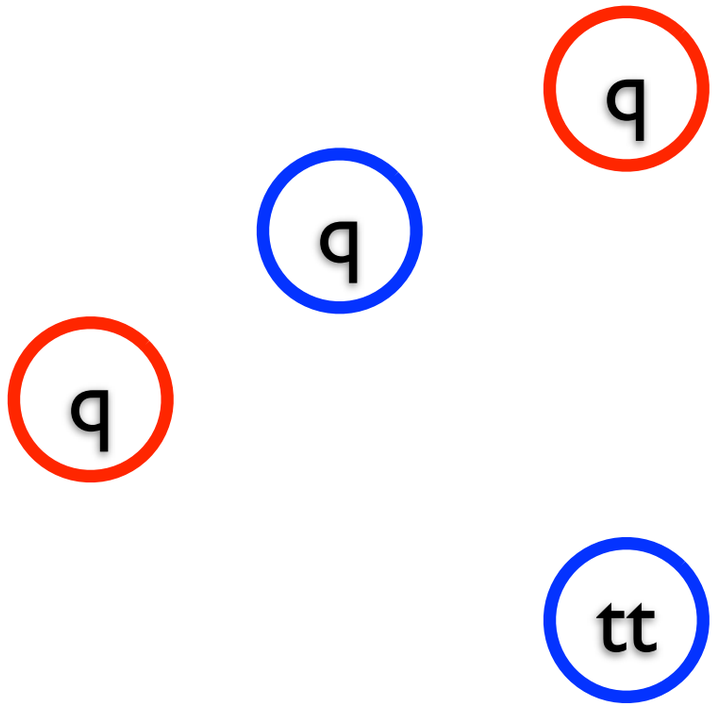
$$\lambda x_1 x_2 ... x_n.\ \text{if } x_i\ M_1 ... M_k \text{ then } N_1 \text{ else } N_2$$

- Does every strategy behave like this?

Early exits

Consider a strategy which plays as shown below:

$$(\mathbf{B} \to \mathbf{B}) \to \mathbf{B}$$



This *does not* correspond to any PCF-definable term.

Bracketing condition

Our normal forms

$$\lambda x_1 x_2 \ldots x_n.\ \text{if } x_i\ M_1 \ldots M_k \text{ then } N_1 \text{ else } N_2$$

and in fact all PCF terms, satisfy a *bracketing condition:*

> no question q is answered until all questions asked after q have been answered

Bracketing condition

- Add a label to moves in arenas: every move is a question or an answer. In a play, an answer move *answers* the question it points to.

- A play *s* satisfies *P-bracketing* if and only if:

  for all prefixes *ta* of s, where *a* is a P-answer, *a* answers the last unanswered question in view(*t*).

Formalities

- An innocent strategy σ is *well-bracketed* if every play $s \in \sigma$ is well-bracketed.
- Composition of well-bracketed strategies yields a well-bracketed strategy.
- The category of arenas and well-bracketed innocent strategies is a CCC; it contains our model of PCF.

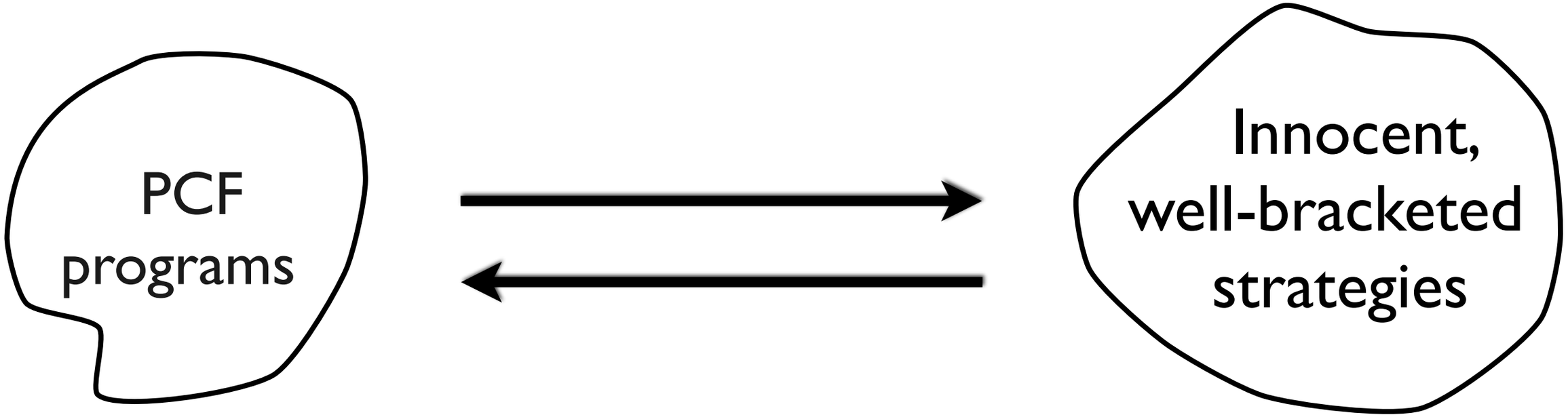
Definability for PCF

Theorem [Hyland-Ong 2000]

Every finite, innocent, well-bracketed strategy corresponds to a term of PCF

- "Finite" means "finite as an E-strategy", i.e. containing a finite number of distinct views.
- The proof is a direct extension of the one for λ-calculus.

Definability for PCF



3. Control

Bracketing vs Control

- The bracketing condition restricts functions to a *stack discipline* for calls and returns.
- In programming terms, this means the absence of control operators.
- Can we make that correspondence precise?

Add a control operator

- Add your favourite control operator to PCF. For example, add an empty type $\bot$ and a constant

$$\text{callcc}: ((\mathbf{B} \to \bot) \to \mathbf{B}) \to \mathbf{B}$$

- How do we model this?

callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

$$((\mathbf{B} \to \bot) \to \mathbf{B}) \to \mathbf{B}$$

callcc

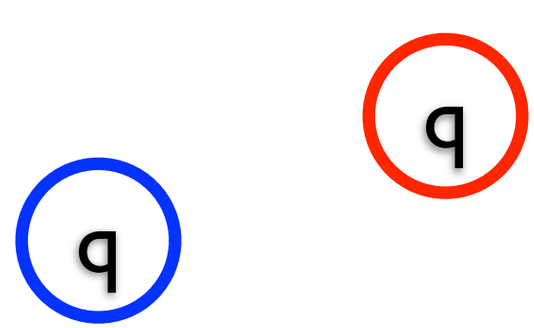
Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

$$((\mathbf{B} \rightarrow \bot) \rightarrow \mathbf{B}) \rightarrow \mathbf{B}$$

callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

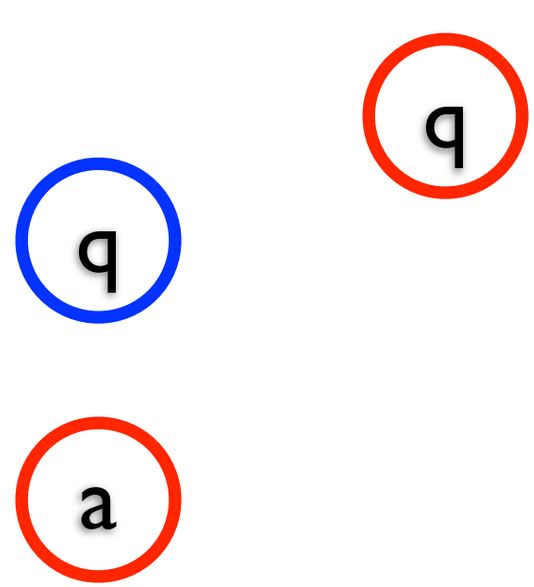
$$((\mathbf{B} \rightarrow \bot) \rightarrow \mathbf{B}) \rightarrow \mathbf{B}$$



callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

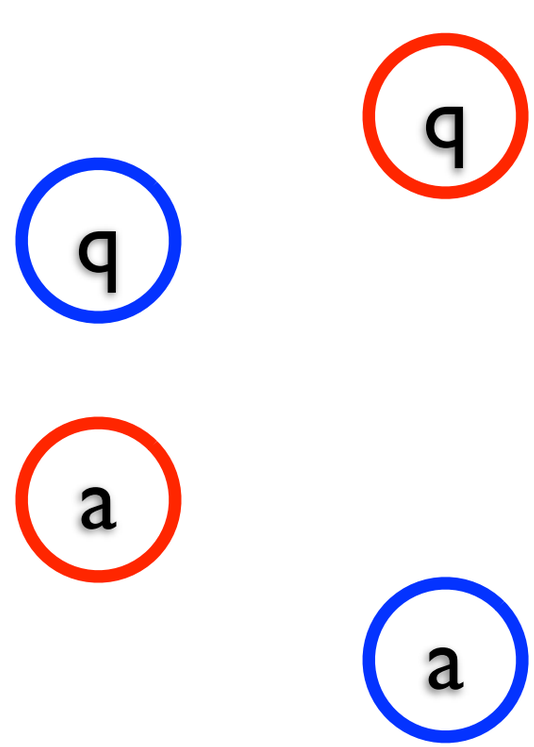
$$((\mathbf{B} \to \bot) \to \mathbf{B}) \to \mathbf{B}$$



callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

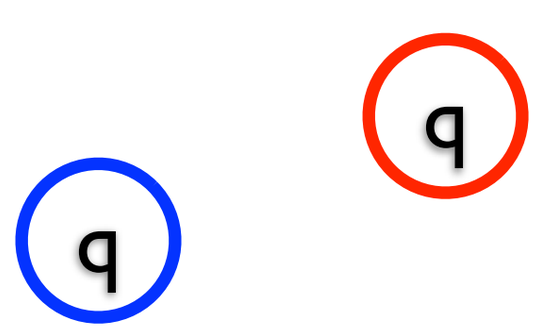
$$((\mathbf{B} \to \bot) \to \mathbf{B}) \to \mathbf{B}$$



callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

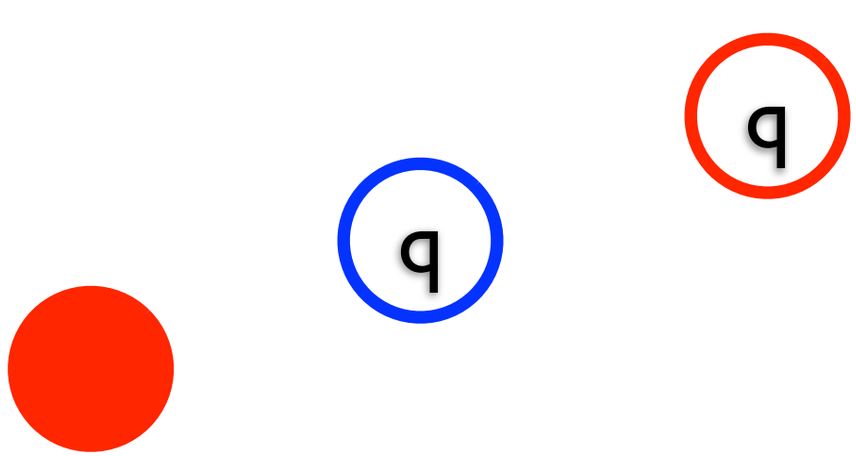
$$((\mathbf{B} \rightarrow \bot) \rightarrow \mathbf{B}) \rightarrow \mathbf{B}$$



callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

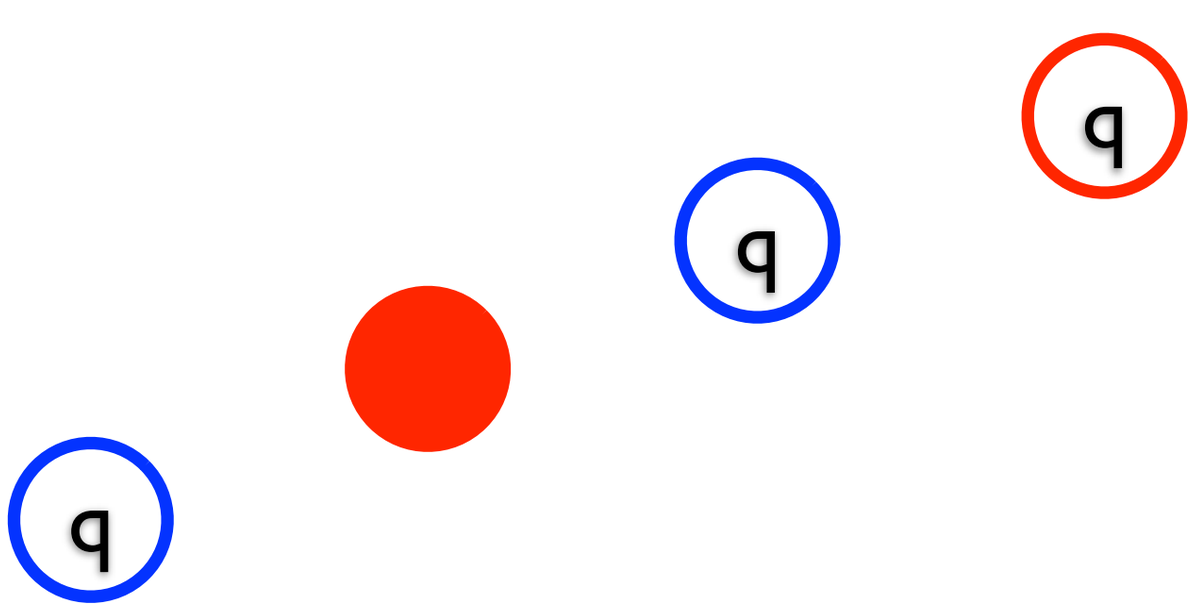
$$((\mathbf{B} \to \bot) \to \mathbf{B}) \to \mathbf{B}$$



callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

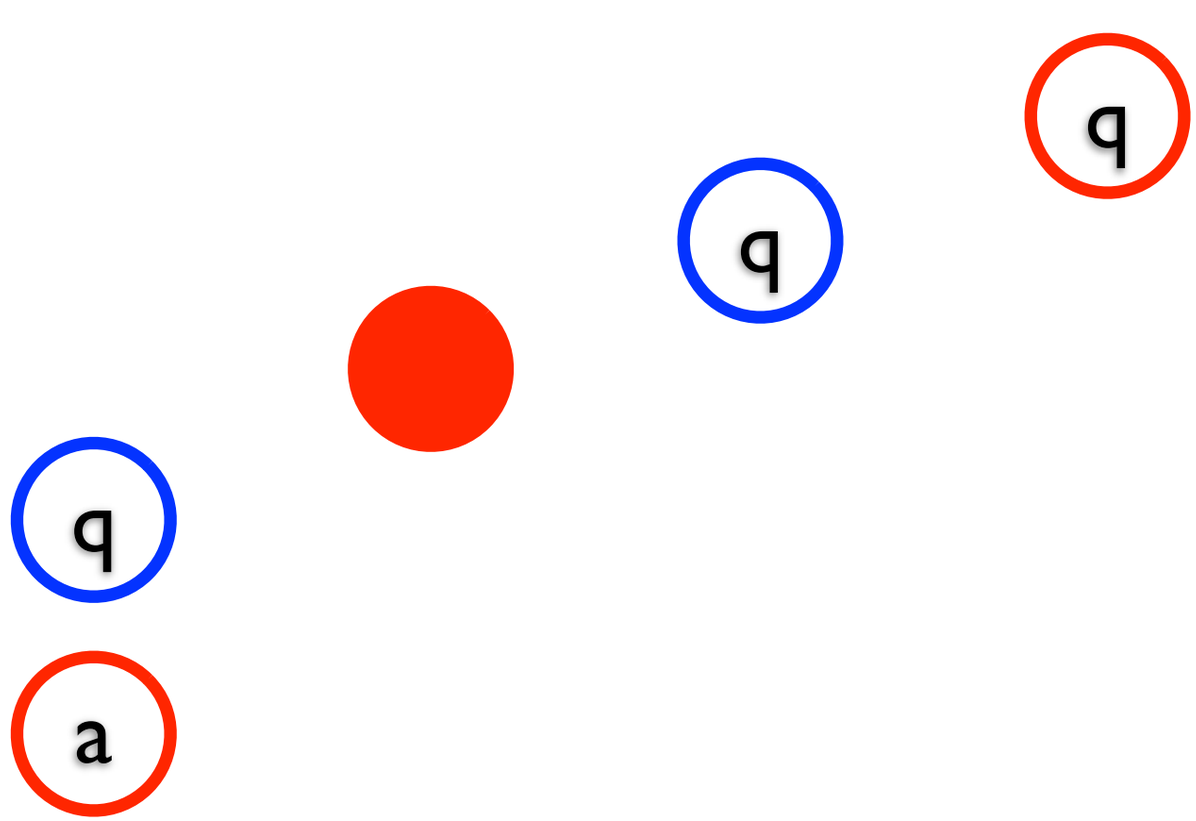
$$((\mathbf{B} \to \bot) \to \mathbf{B}) \to \mathbf{B}$$



callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

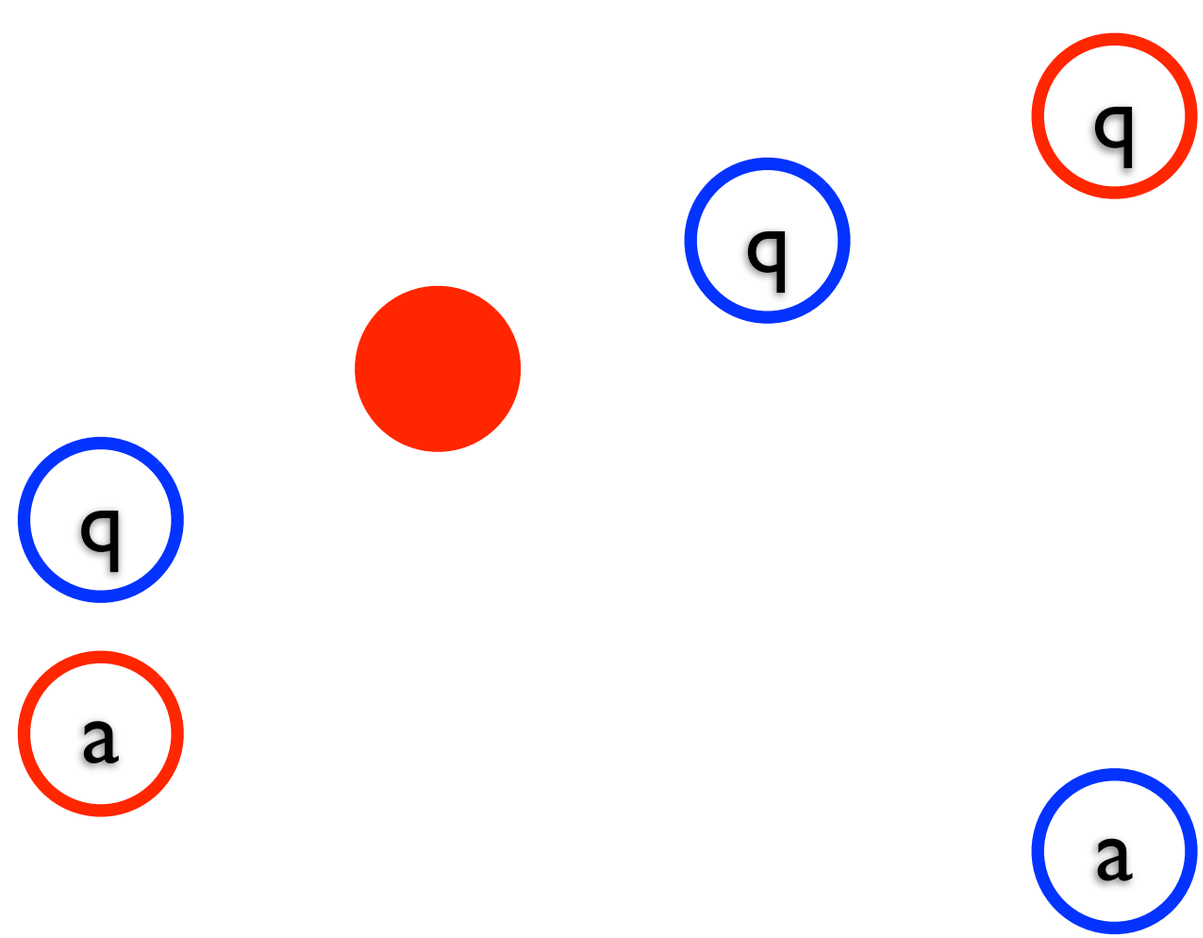
$$((\mathbf{B} \to \bot) \to \mathbf{B}) \to \mathbf{B}$$



callcc

Interpreting ⊥ as the one-move arena, we model callcc with the following strategy:

$$((\mathbf{B} \rightarrow \bot) \rightarrow \mathbf{B}) \rightarrow \mathbf{B}$$



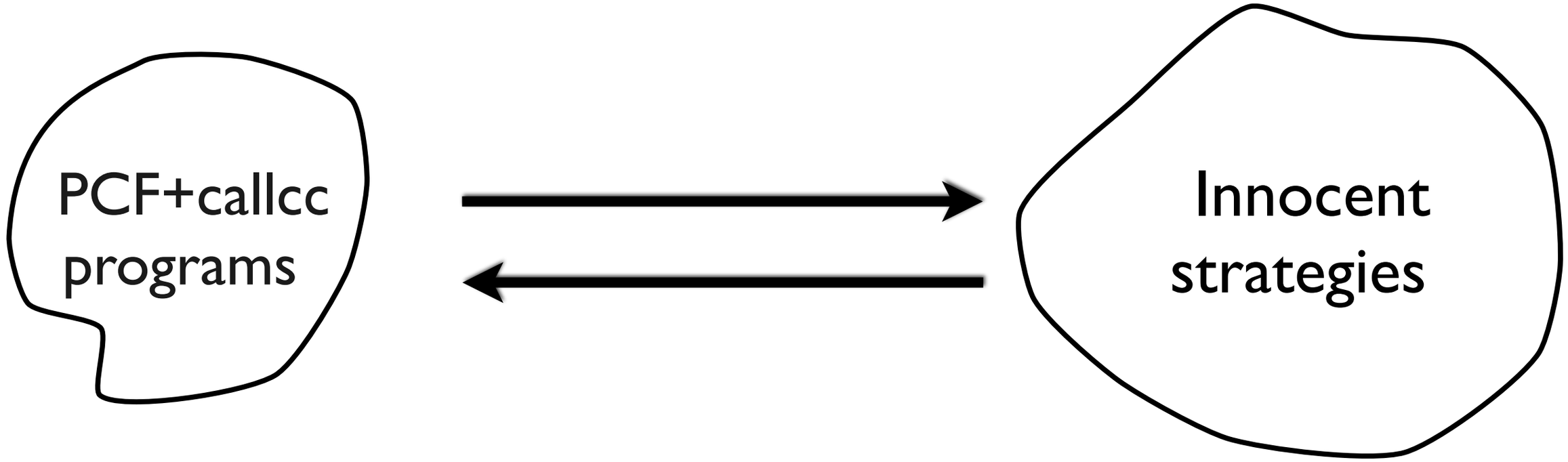
Definability

This gives us a model of PCF + callcc, with a definability property.

Theorem [Laird 1997]

Every finite, innocent, (not necessarily well-bracketed) strategy corresponds to a term of PCF + callcc.

Definability for PCF +callcc



Games and linear continuations

- This result demonstrates that the games model is a continuations-based model, slightly disguised.

- Laird (2005) shows that the well-bracketed strategies are exactly those obeying a certain *linear continuation passing* regime:

  - answers are continuations that can be invoked at most once.

4. Beyond innocence: state

Non-innocent strategies

- Strategies without the innocence constraint form another CCC.
- Our analysis so far has exploited a tight correspondence between *views* and *paths in syntax trees*.
- If we abandon innocence, what do we get?

Non-innocence

$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$

λF.λx.

F

λy.

y

λy.

F

Non-innocence

$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$

λF.λx.

F

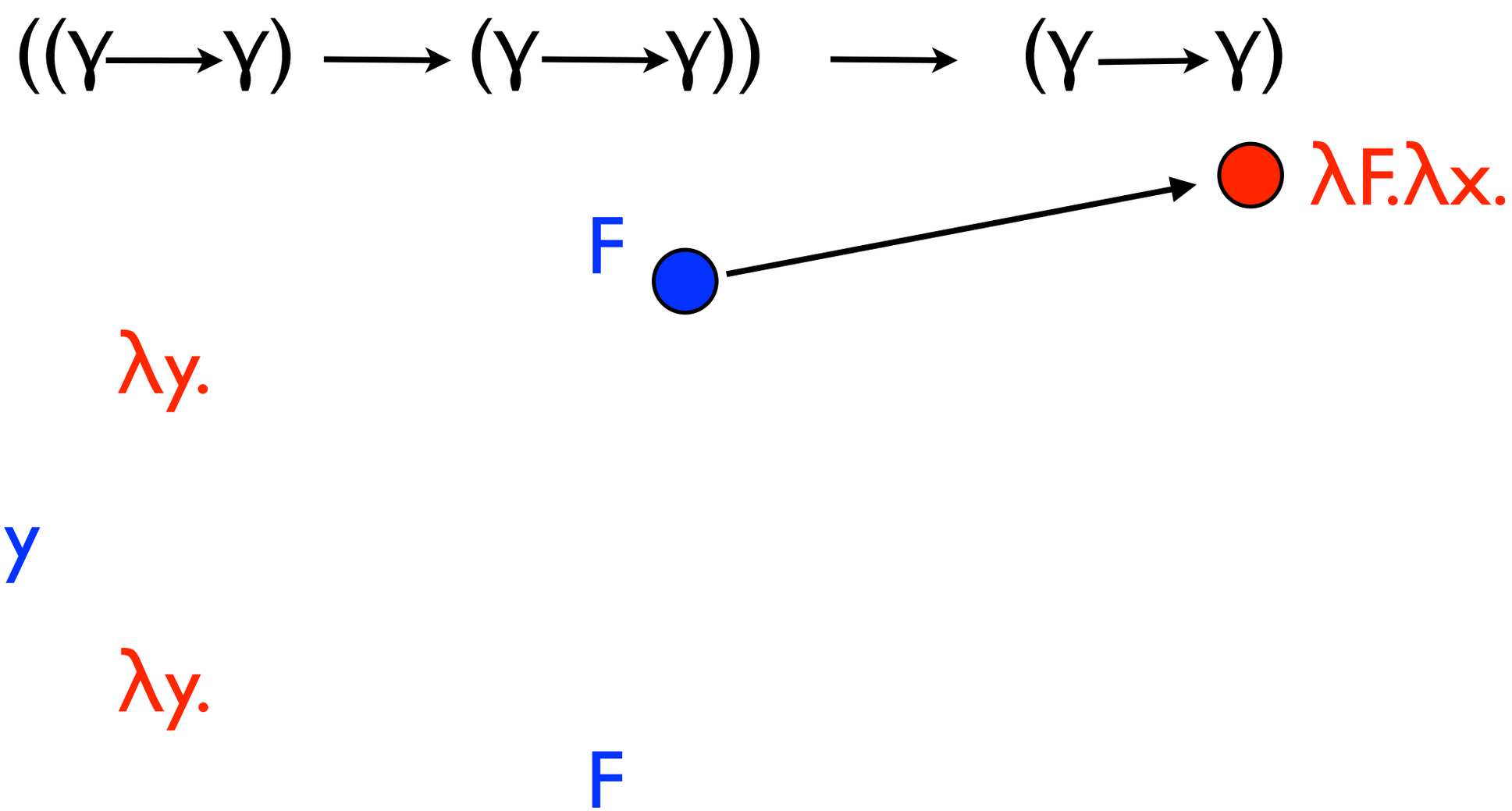
λy.

y

λy.

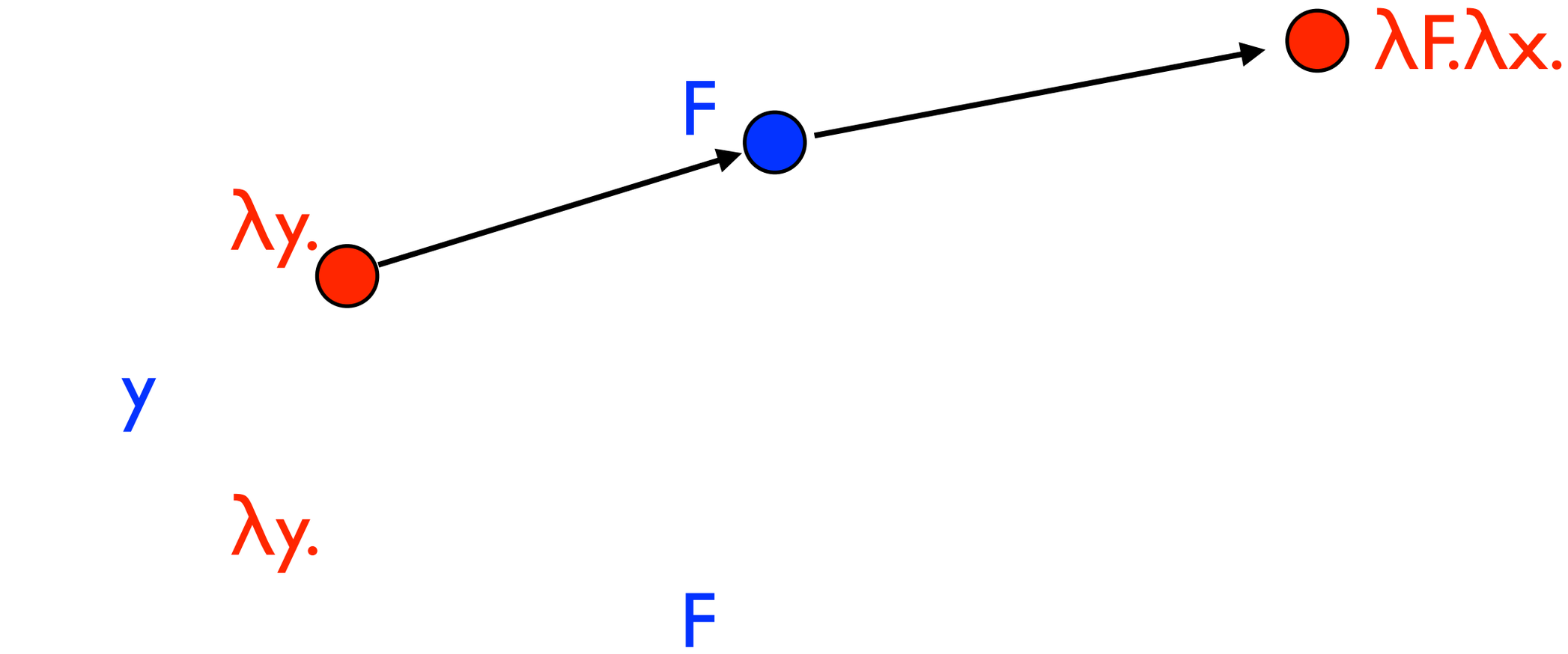
F

Non-innocence



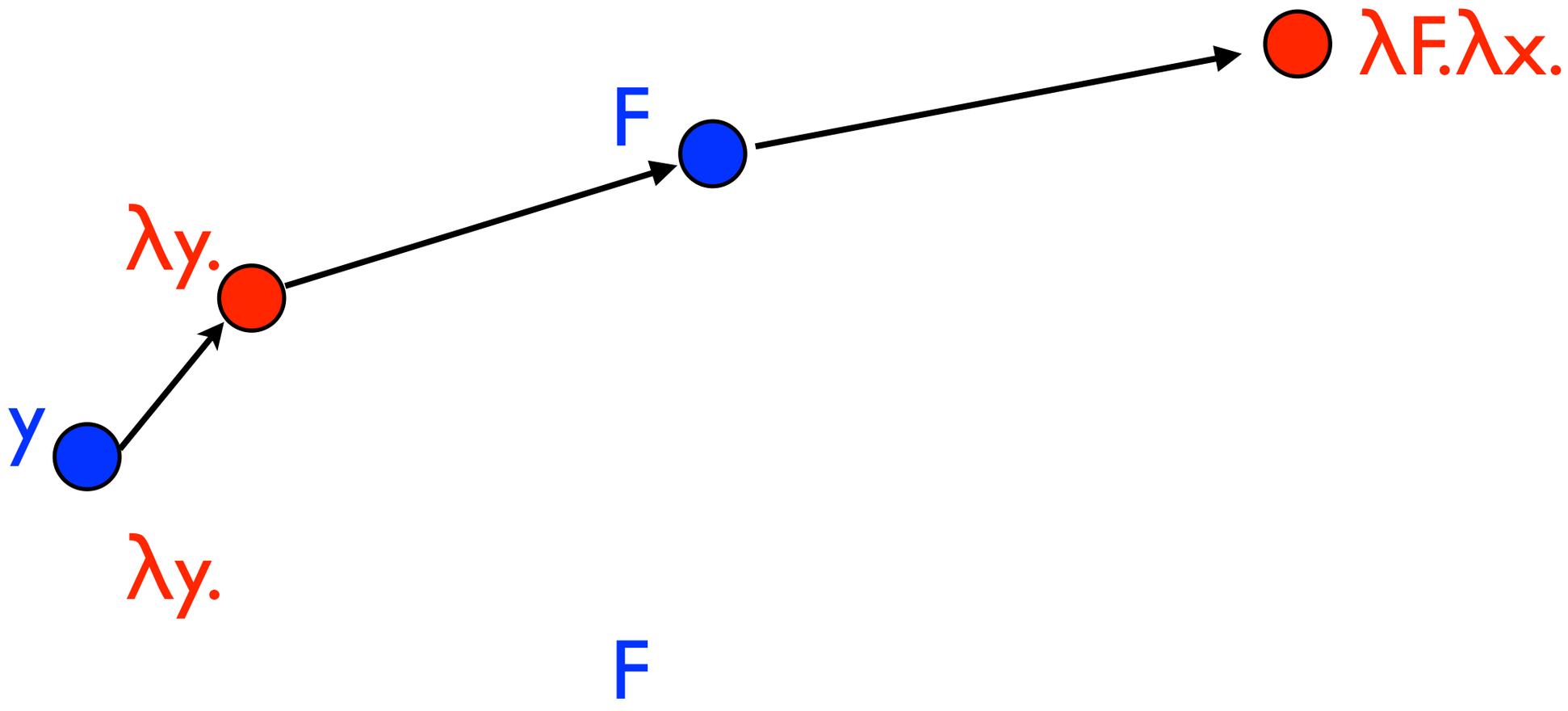
Non-innocence

$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$



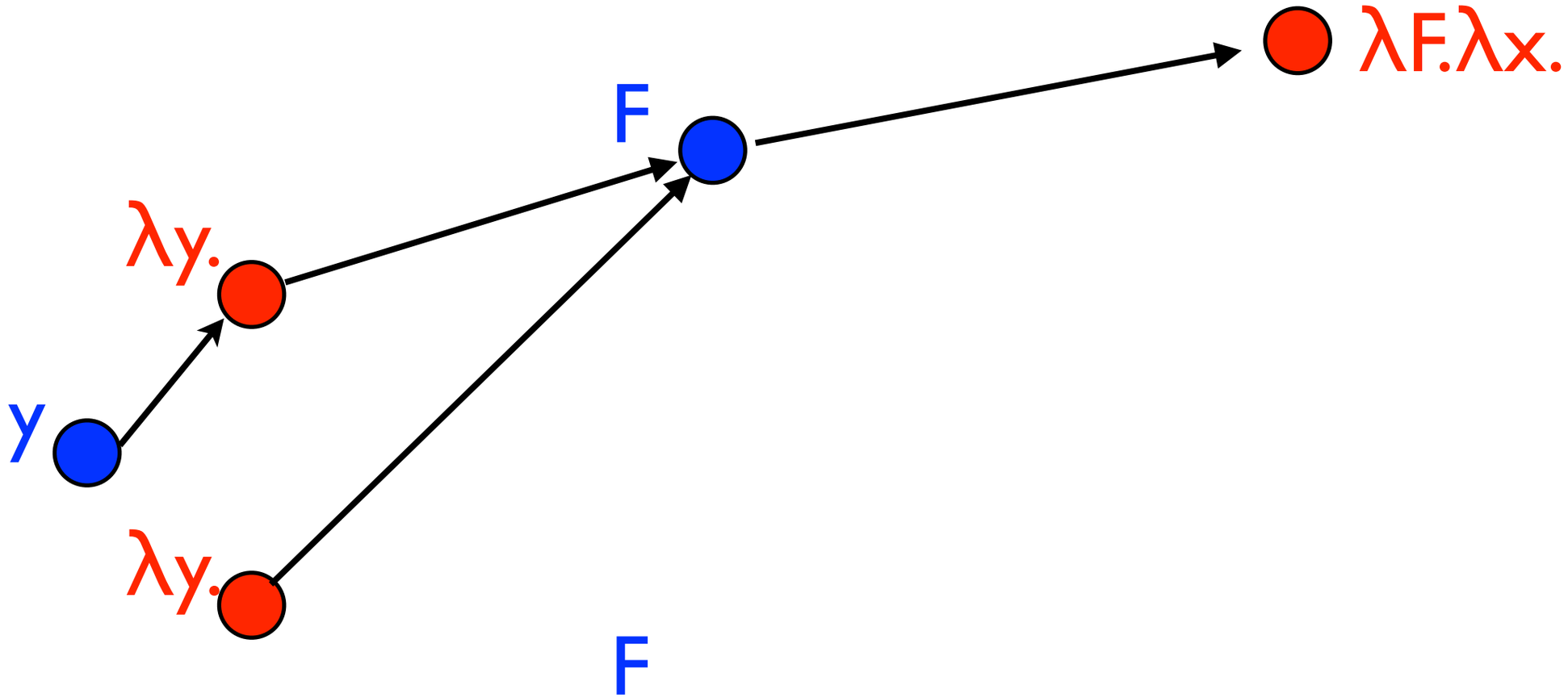
Non-innocence

$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$

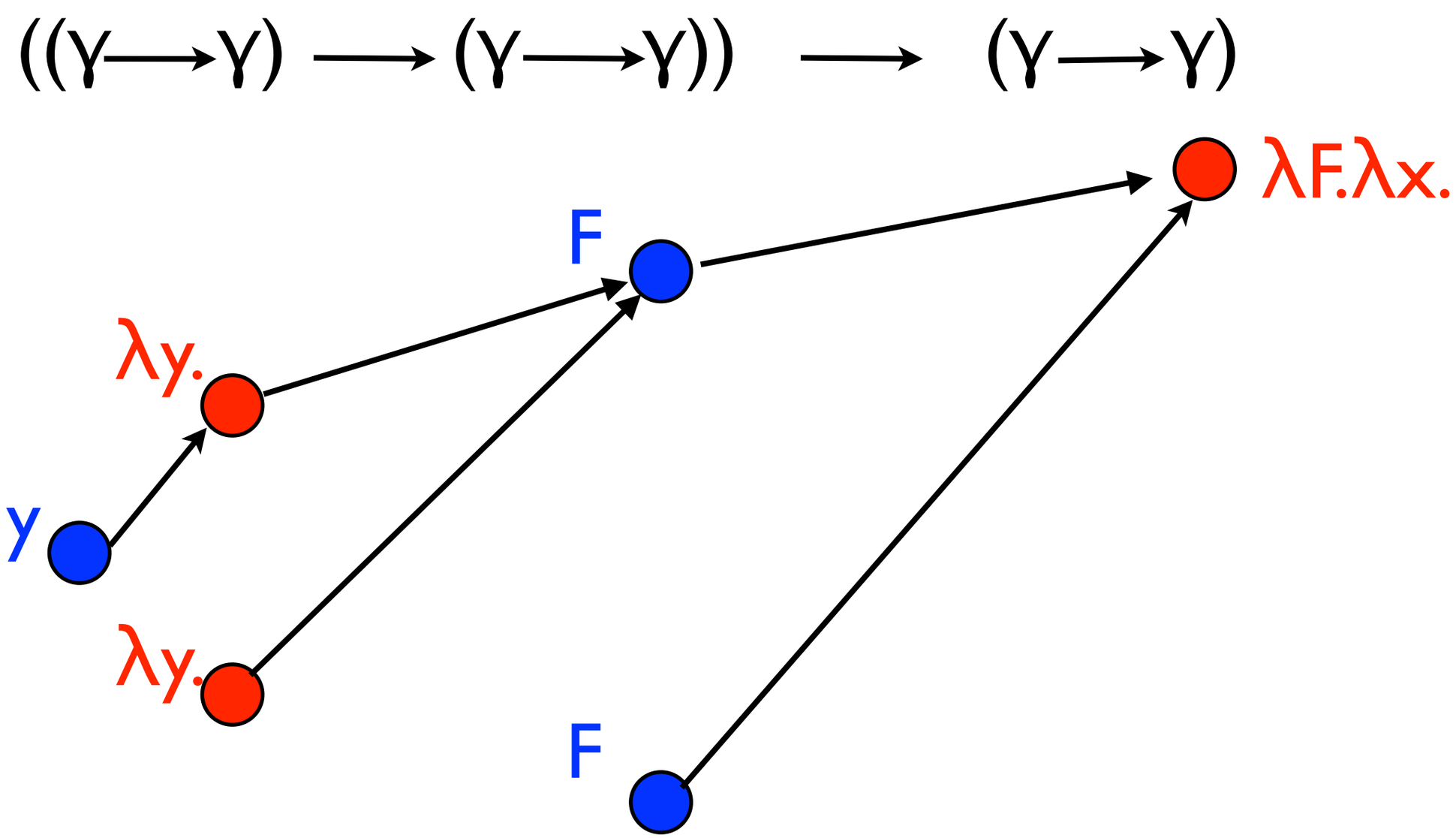


Non-innocence

$$((\gamma \to \gamma) \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$$



Non-innocence



Changing minds

- How can a program change its responses like this?
- Using state!

Changing minds

- How can a program change its responses like this?
- Using state!

```
λF.λx.new v:= true in
        F(λy. if !v then v:=false; return y
              else F( ... )  )
```

Stateful strategies

- Non-innocent strategies *directly* represent stateful computation.
- The state itself is *implicit*: what we see in the strategy is the *behaviour* implemented by using the state.
- Contrast with explicit-state models, e.g. those based on a state monad.

Stateful strategies

- Non-innocent strategies *directly* represent stateful computation.
- The state itself is *implicit*: what we see in the strategy is the *behaviour* implemented by using the state.
- Contrast with explicit-sta
  those based on a state mona

this idea was first explored by Uday Reddy (1993)

Idealised Algol

- Reynolds's Idealised Algol is a prototypical higher-order imperative programming language

IA = PCF + assignable variables + block structure

Idealised Algol

- Reynolds's Idealised Algol is a prototypical higher-order imperative programming language

IA = simple while
programs + block
structure + λ-calculus

Idealised Algol

- Types: A :: = comm | exp | var | A → A
- Terms: M ::= PCF | x:= M | !x | M ; M
  | new x in M

Commands and variables

- To interpret commands and variables, we can no longer rely on the analogy with normal-form trees.
- We have to do some semantics!
- Consider the observable actions available for each type, and build an appropriate arena.

Commands

Commands

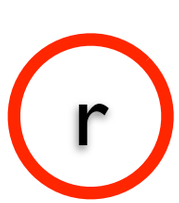
- With no explicit store, what can we observe of a command?

Commands

- With no explicit store, what can we observe of a command?
  - We can try to run it: "run"

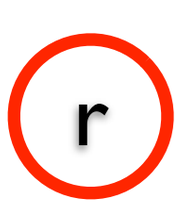
Commands

- With no explicit store, what can we observe of a command?
- We can try to run it: "run"



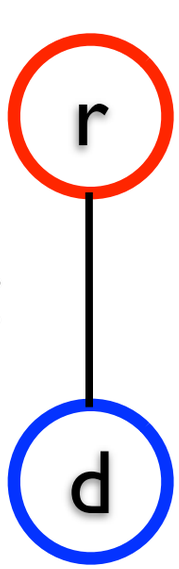
Commands

- With no explicit store, what can we observe of a command?
- We can try to run it: "run"
- We will notice when it terminates: "done".



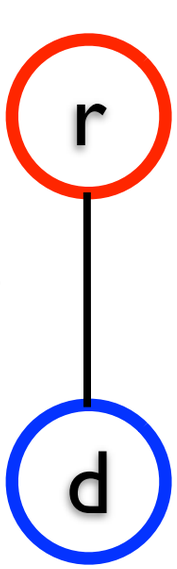
Commands

- With no explicit store, what can we observe of a command?
- We can try to run it: "run"
- We will notice when it terminates: "done".



Commands

- With no explicit store, what can we observe of a command?
- We can try to run it: "run"
- We will notice when it terminates: "done".
- That's all.



Sequential composition

comm → comm → comm

r

r

d

r

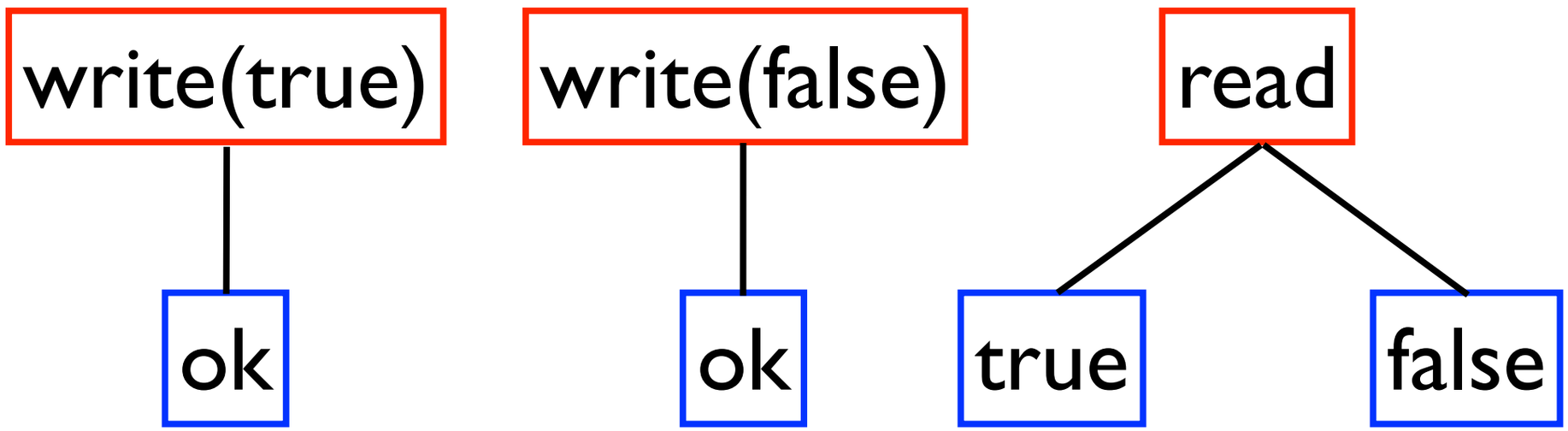
d

d

Variables

- Variables are more complex:
  - we can try to read a value, and get something back
  - we can try to store a value; this is like a command, so we just observe termination.

Variables



Assignment

var → exp → comm

r

q

a

write(a)

ok

d

Example

x: var, y: var ⊢ x := not (!y) : comm

Example

x: var, y: var ⊢ x := not (!y) : comm

r

Example

x: var, y: var ⊢ x := not (!y) : comm

r

read

Example

x: var, y: var ⊢ x := not (!y) : comm

r

read

b

Example

x: var, y: var ⊢ x := not (!y) : comm

r

read

b

write(not b)

Example

x: var, y: var ⊢ x := not (!y) : comm

r

read

b

write(not b)

ok

Example

x: var, y: var ⊢ x := not (!y) : comm

r

read

b

write(not b)

ok

d

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

ok

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

ok

r

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

ok

r

d

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

ok

r

d

read

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

ok

r

d

read

b

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

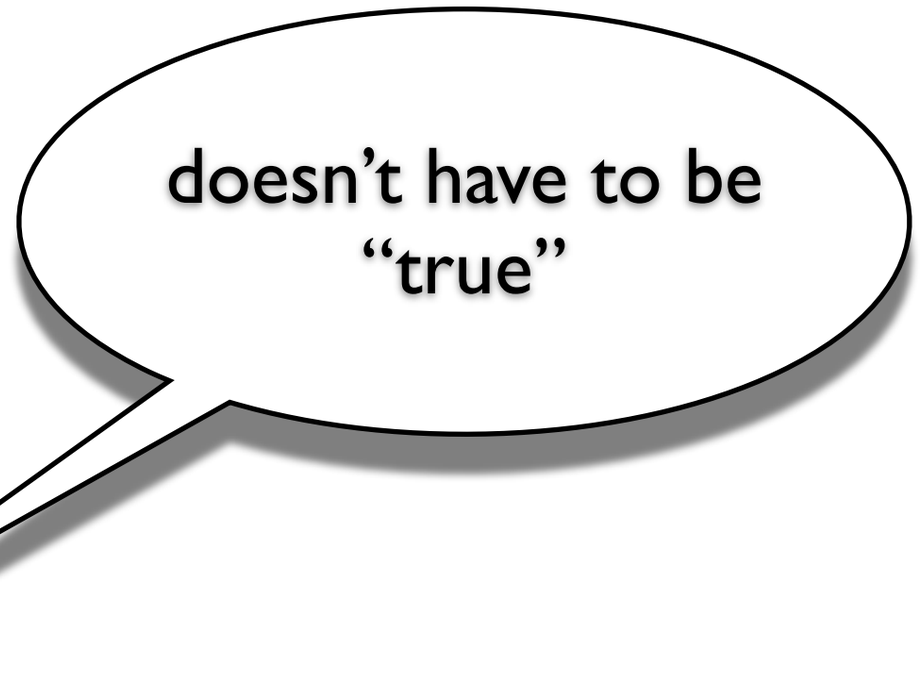
ok

r

d

read

b



Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

ok

r

d

read

b

write(not b)

doesn't have to be "true"

Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

r

write(true)

ok

r

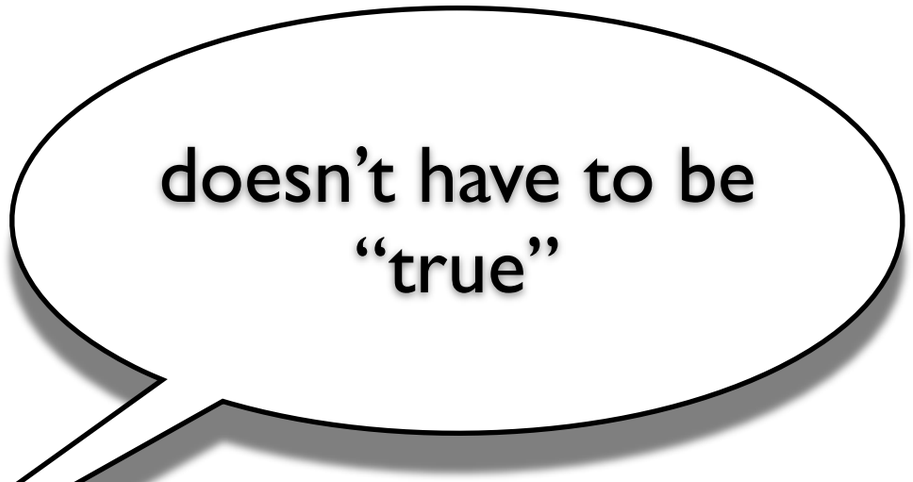
d

read

b

write(not b)

ok



Another example

c:comm, x: var ⊢ x:=true; c; x := not(!x) : comm

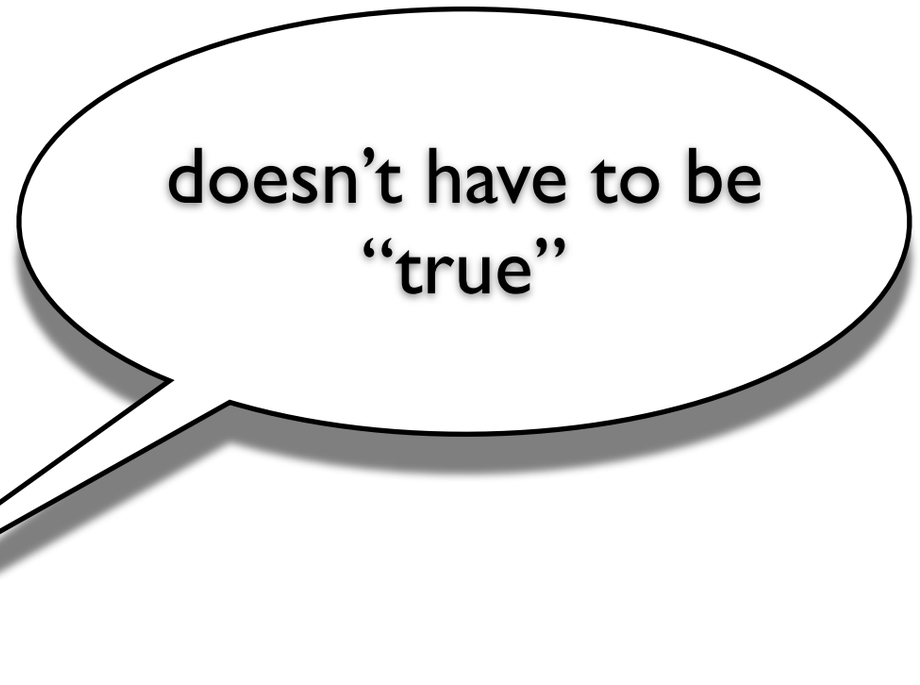
r

write(true)

ok

r

d



read

b

write(not b)

ok

d

Bad variable behaviour

Bad variable behaviour

- Why don't we try to restrict plays so that the *b* in the previous example is always "true"?

Bad variable behaviour

- Why don't we try to restrict plays so that the *b* in the previous example is always "true"?
- The command c could later become bound to something that alters x, so it is *vital* to allow the *read* to return any value.

Local variable behaviour

- If we wrap the term in a variable allocation

  new x in x:=true; c; x := not(!x)

  it is no longer possible for c to alter x.

- The variable x becomes a *good variable*.
- It also becomes *hidden*: the environment should not know it is there.

Variable allocation

- The command *new x in M* is just like *M*, but:
  - *x* is bound to a storage cell, so *M*'s interactions with *x* should be variable-like.
  - the outside world should no longer see *x*.
- How can we model this? Interaction plus hiding!

A cell strategy

(var → comm) → comm

r

r

⋮

s

⋮

d

d

A cell strategy

(var → comm) → comm

r

r

⋮

s

⋮

d

d

any “good” sequence of reads and writes

Allocation in action

(var → comm) → comm

r

r

write(true)

ok

read

true

write(false)

ok

d

d

Allocation in action

(var → comm) → comm

r

r

write(true)

ok

answering "true" here requires non-innocence

read

true

write(false)

ok

d

d

Allocation in action

comm → (var → comm)

r

write(true)

ok

r

d

read

true

write(false)

ok

d

Allocation in action

comm → (var → comm) → comm

r

r

write(true)

ok

r

d

read

true

write(false)

ok

d

d

Allocation in action

comm → comm

r

r

d

d

Allocation in action

comm → comm

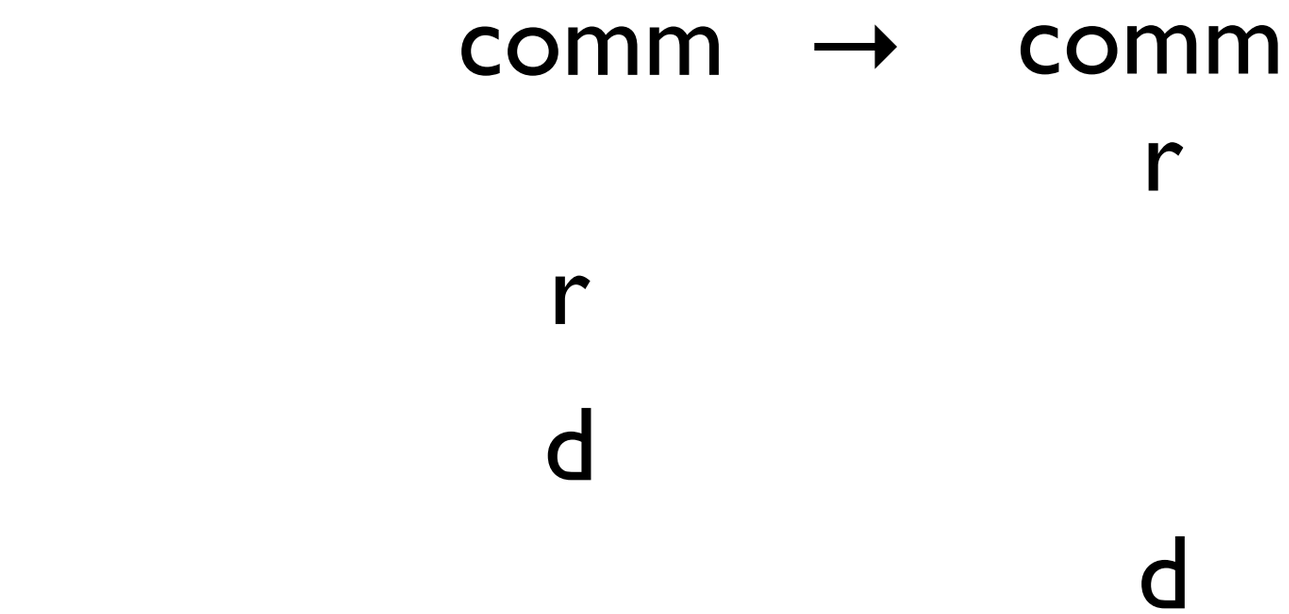
r

r

d

d

Allocation in action



The internal interaction went as expected,
and now it is invisible.

A remark

- In this model, *only* variable allocation requires a non-innocent strategy: terms without *new x in* ... are interpreted innocently.
- This is a marked difference from explicit state models, where assignment and lookup operations access the state.

Definability?

- We have a model of Idealised Algol, and we can prove that it is sound.
- Does it have the definability property?

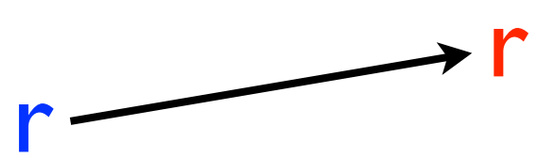
Visibility

- Our non-innocent strategies are very powerful. E.g.:
  (A → (A → comm) →comm) → comm

r

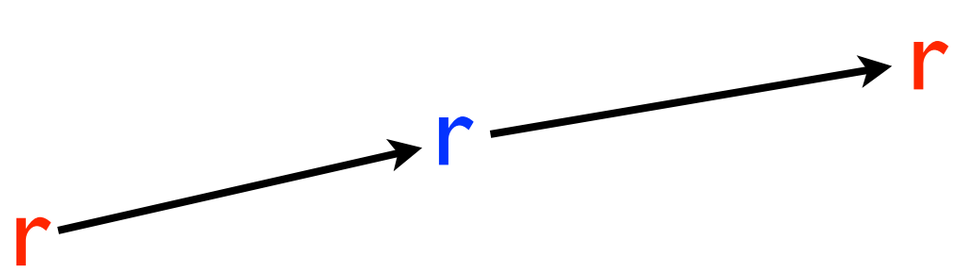
Visibility

- Our non-innocent strategies are very powerful. E.g.:
  (A → (A → comm) →comm) → comm



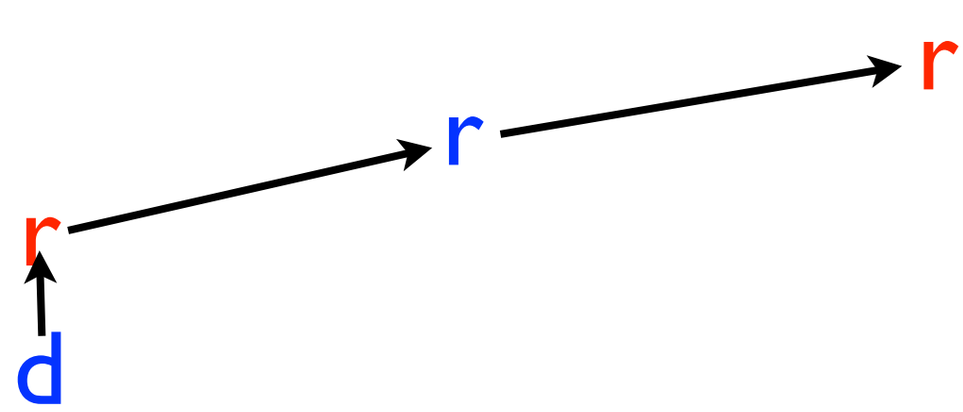
Visibility

- Our non-innocent strategies are very powerful. E.g.:
  (A → (A → comm) →comm) → comm



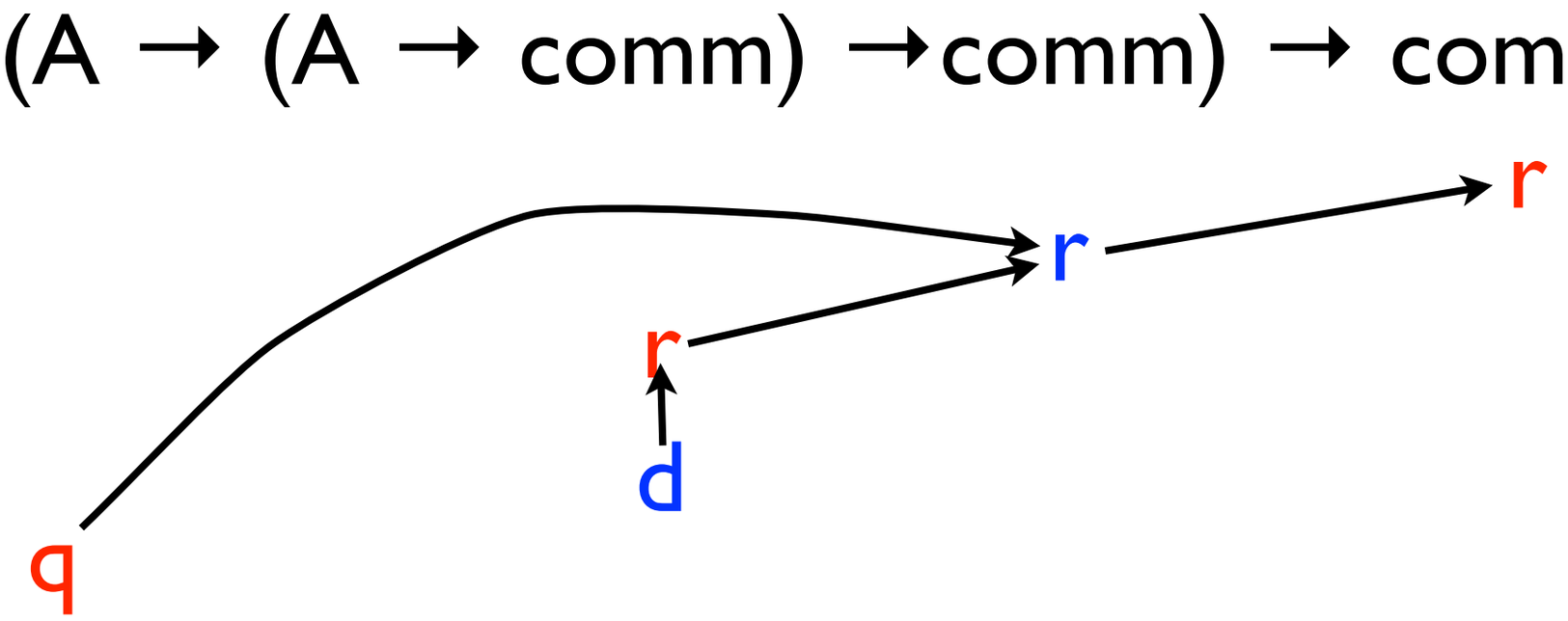
Visibility

- Our non-innocent strategies are very powerful. E.g.:
  (A → (A → comm) →comm) → comm



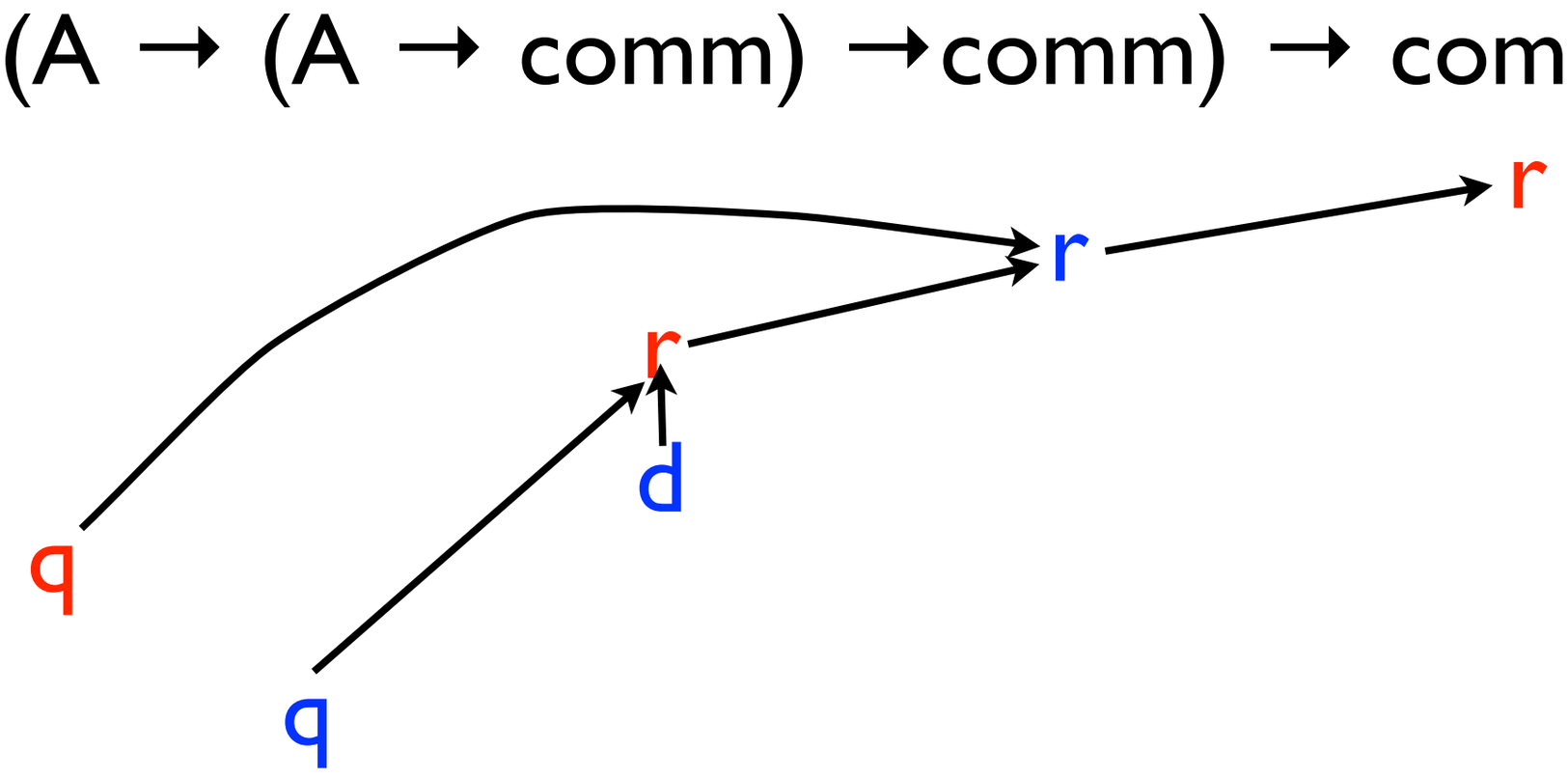
Visibility

- Our non-innocent strategies are very powerful. E.g.:
  (A → (A → comm) →comm) → comm



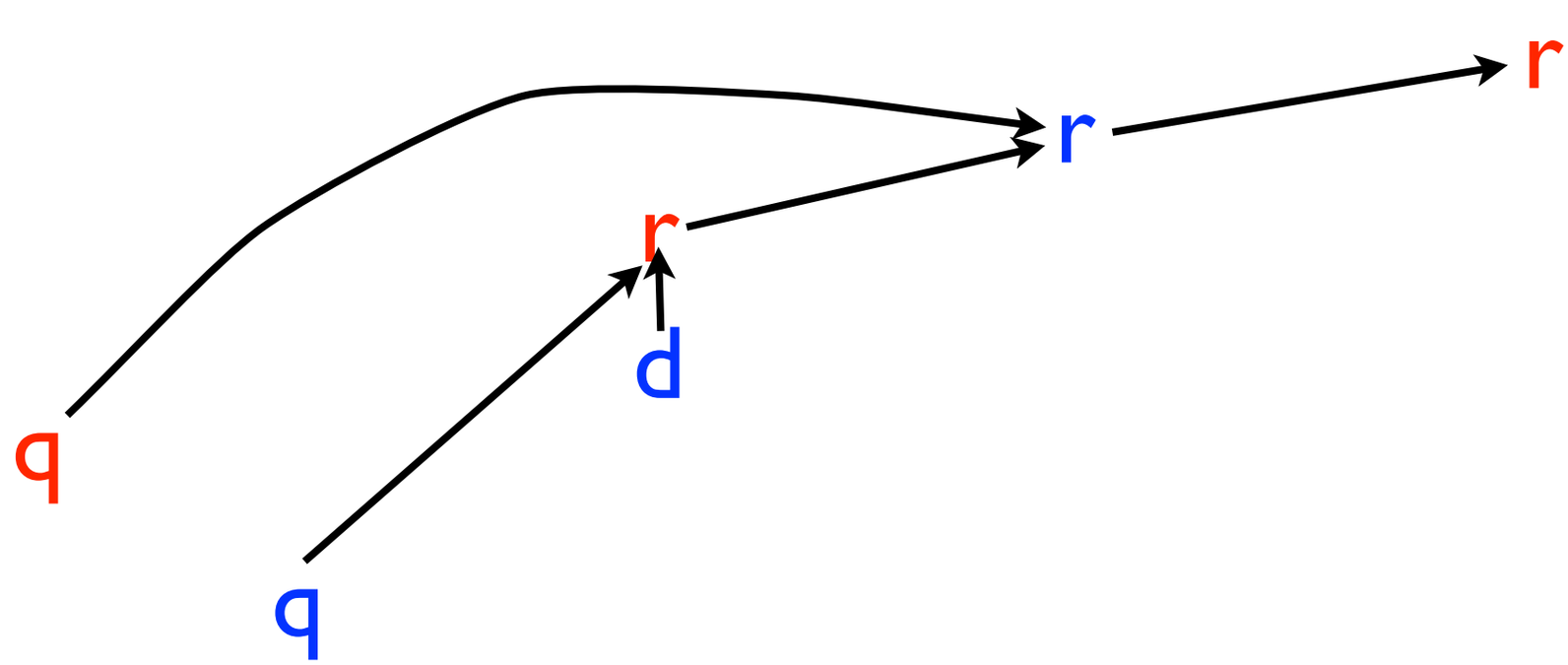
Visibility

- Our non-innocent strategies are very powerful. E.g.:
  (A → (A → comm) →comm) → comm



Visibility

- Our non-innocent strategies are very powerful. E.g.:
  (A → (A → comm) →comm) → comm



- No Idealised Algol program has this behaviour.

Visibility

- Our non-innocent strategies are very powerful. E.g.:

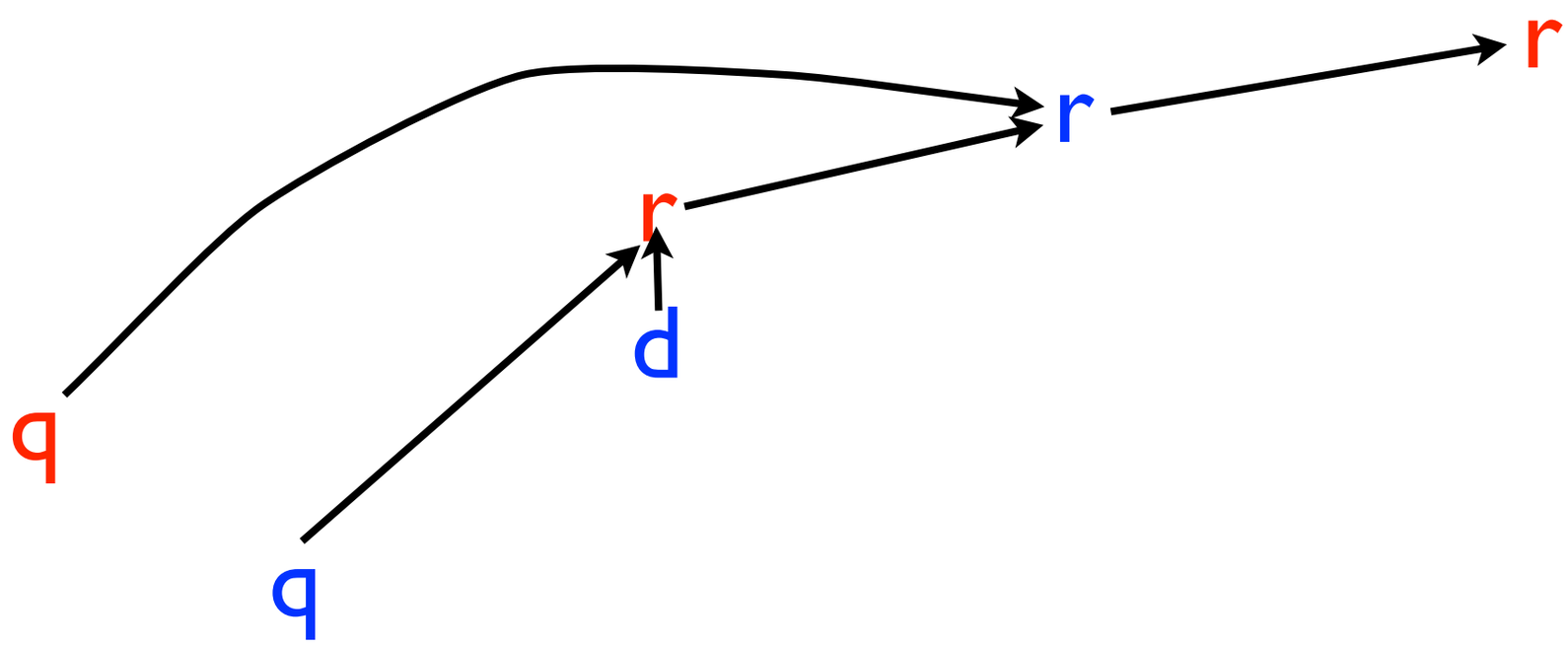
- No Idealised Algol program has this behaviour.

Visibility

- To eliminate this strategy, we impose a new constraint: *visibility*.
- A play *s* satisfies *P-visibility* if, for every prefix *tm* where *m* is a P-move, the justifier of *m* is in view(*t*).
- (All plays in innocent strategies satisfy this automatically.)

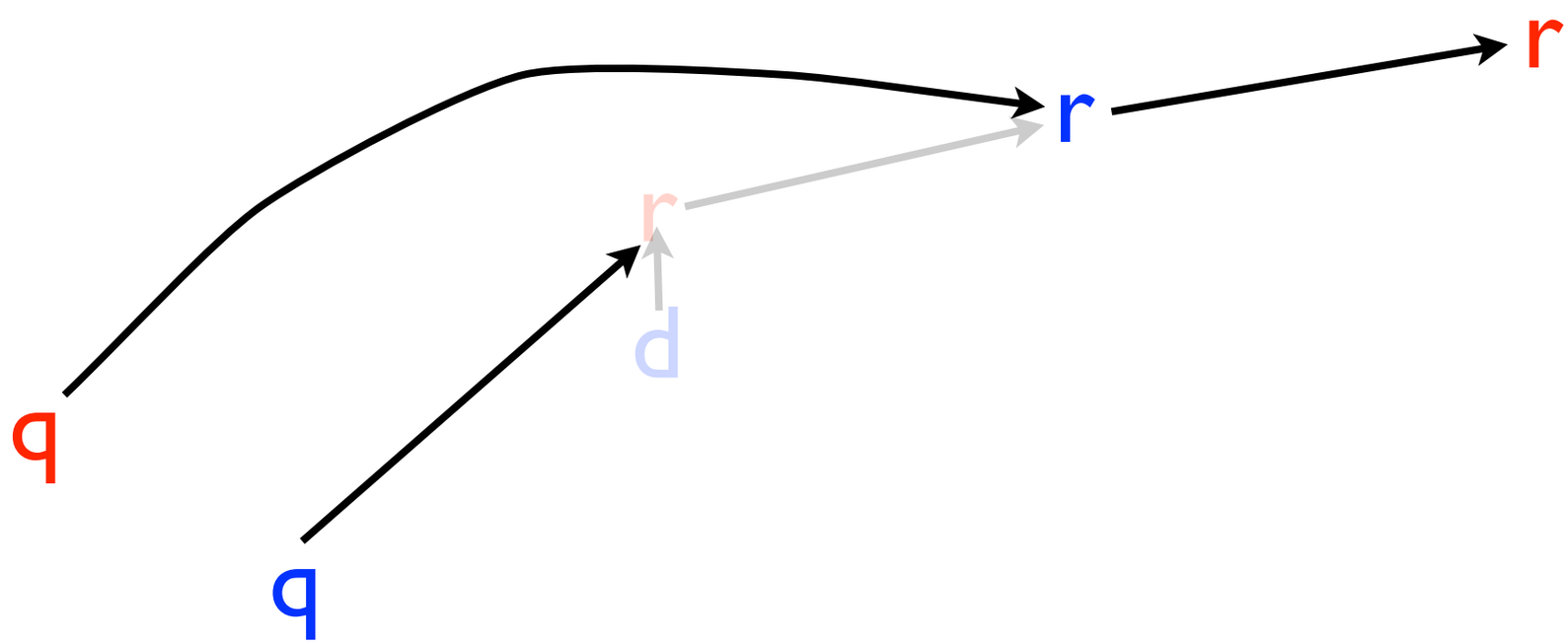
Visibility

(A → (A → comm) → comm) → comm



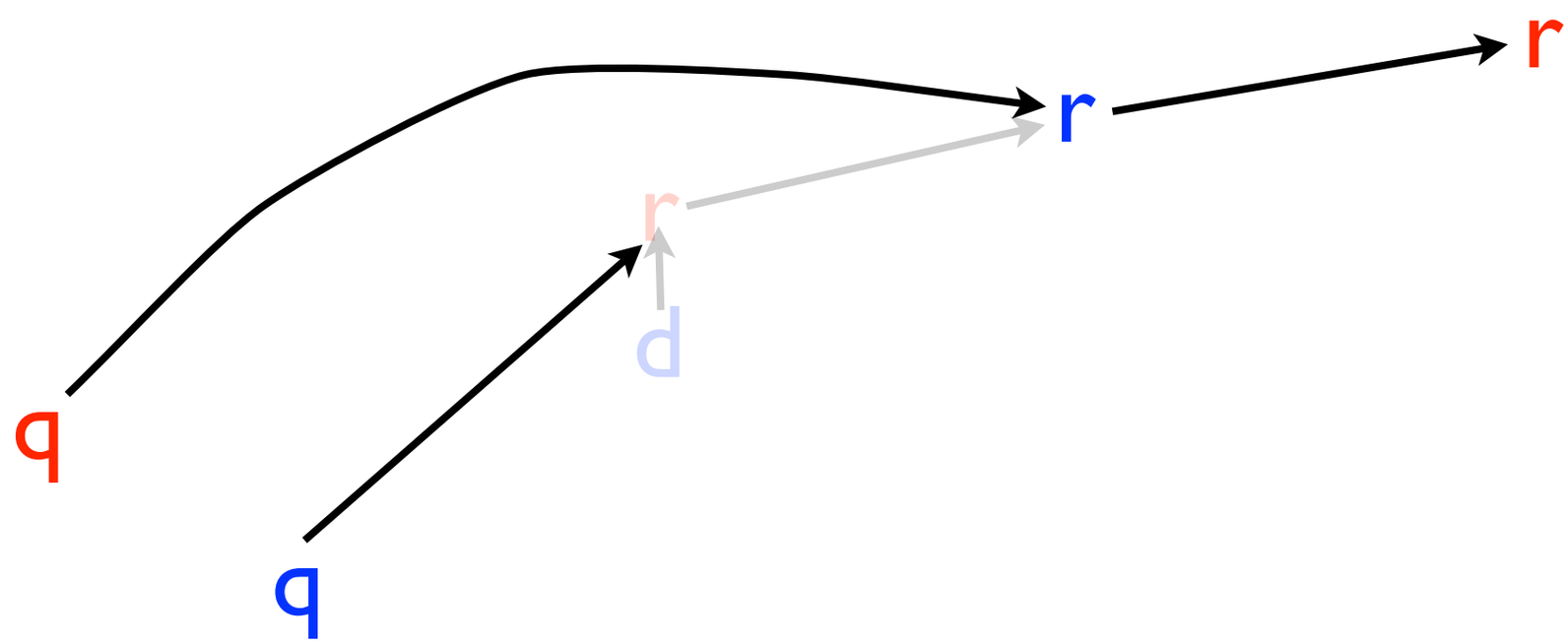
Visibility

(A → (A → comm) →comm) → comm



Visibility

(A → (A → comm) → comm) → comm



Eliminating such plays lets us recover definability.
But how to prove it?

Proving definability

- We cannot directly use the same approach as in the innocent case: strategies no longer correspond to λ-terms.
- We can *reduce* the problem to the innocent case.

Factorization

Theorem [Abramsky + M 1996]
Every well-bracketed, P-visible strategy on $A \to$ comm is of the form σ ; cell for some *innocent* σ : $A \to (\text{var} \to \text{comm})$.

Factorization

Theorem [Abramsky + M 1996]
Every well-bracketed, P-visible strategy on $A \to comm$ is of the form σ ; cell for some *innocent* σ : $A \to (var \to comm)$.

Proof: innocently simulate the non-innocent strategy by storing the history in the cell.

Factorization

Theorem [Abramsky + M 1996]
Every well-bracketed, P-visible strategy on $A \to$ comm is of the form σ ; cell for some *innocent* σ : $A \to (\text{var} \to \text{comm})$.

- Definability for innocent strategies works as usual.

Factorization

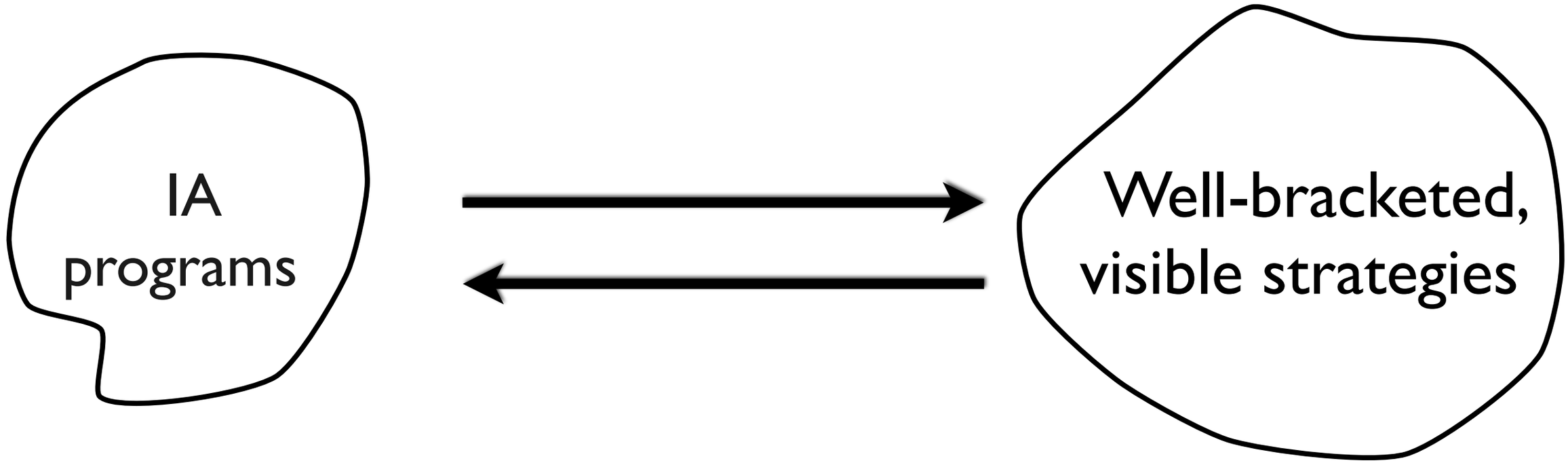
Theorem [Abramsky + M 1996]
Every well-bracketed, P-visible strategy on $A \to \mathrm{comm}$ is of the form σ ; cell for some *innocent* σ : $A \to (\mathrm{var} \to \mathrm{comm})$.

- Definability for innocent strategies works as usual.
- If M is a term defining σ, then *new x in M* defines σ ; cell.

Another definability result

Theorem [Abramsky + M 1996] Every finite, well-bracketed, P-visible strategy on A → comm is definable by a term of Idealised Algol.

Definability for Idealised Algol



Another definability result?

Probably a theorem that I don't think anyone has actually bothered to prove

Every finite (not necessarily well-bracketed) P-visible strategy on A → comm is definable by a term of Idealised Algol + callcc.

5. Beyond visibility: more state

What can general strategies express?

- I am afraid I can no longer pretend to make the progression seem logical: I'll just have to tell you.
- Strategies that may break visibility correspond exactly to programs with *higher order store*.

Ground vs higher order store

- Because ground-type data can be completely evaluated, assignment was easy to interpret.

var → exp → comm

r

q

a

write(a)

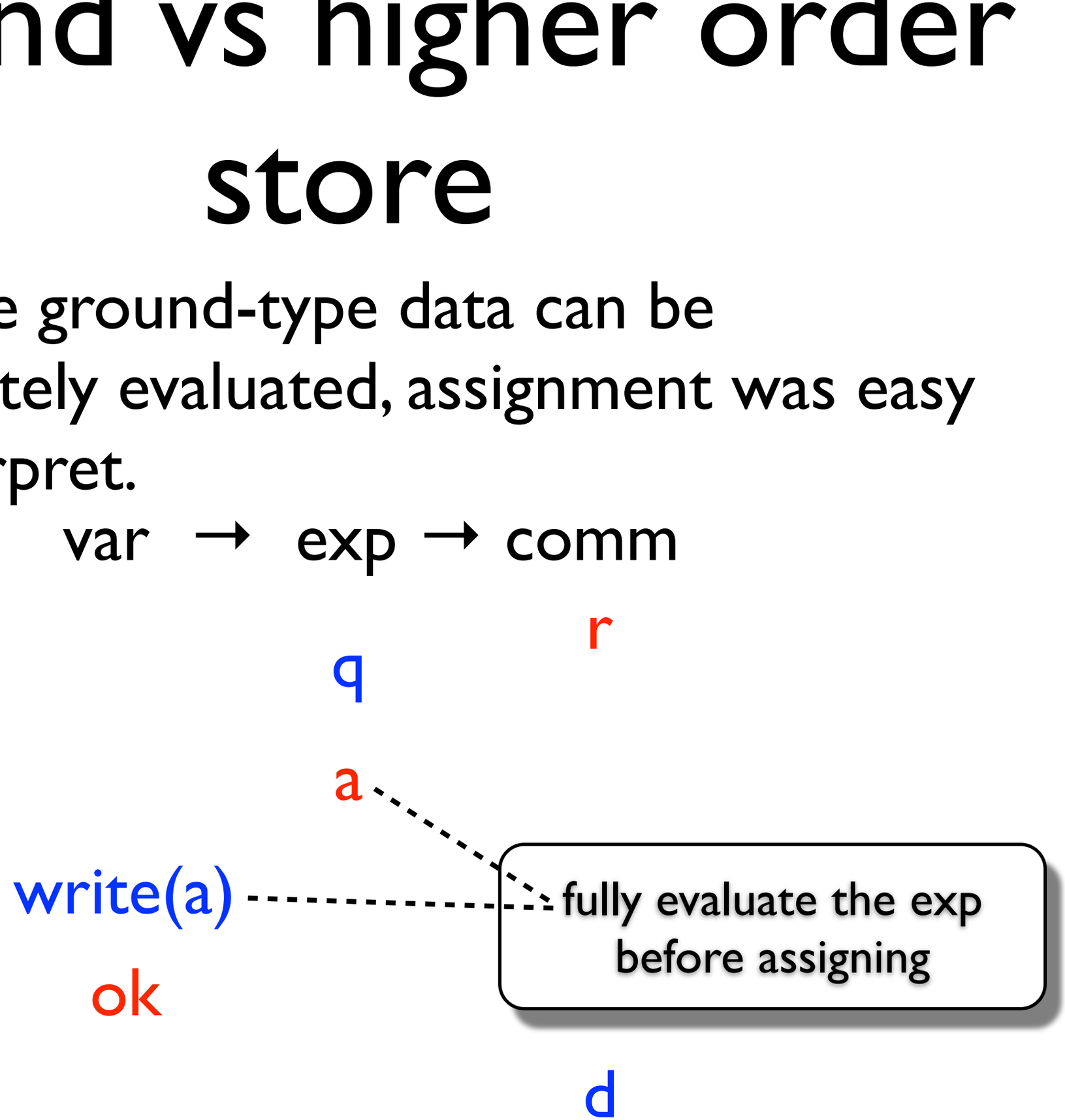
ok

d

Ground vs higher order store

- Because ground-type data can be completely evaluated, assignment was easy to interpret.

var → exp → comm



Ground vs higher order store

- How could we do this for a general type *A*?

var[A] → A → comm

r

q

q’

Ground vs higher order store

- How could we do this for a general type *A*?

var[A] → A → comm

r

q

q’

this is a non-final O-move; what on earth should we do now?

What is var[A] anyway?

- How can we store a program of type A in a variable?

- We can hardly have a move

$$\text{write}(\sigma)$$

where σ is a strategy.

var take two

- An alternate var type, for ground data, is given by

$$\mathrm{exp} \times (\mathrm{exp} \to \mathrm{comm})$$

- Think of this as the product of
  - the "read method", returning an exp
  - the "write method", taking an exp and returning the assignment command

Definability again

- This gives another way to interpret Idealised Algol, and we can obtain another definability result.
- Perhaps this version of var will generalise to arbitrary types? Define

$$\mathrm{var}[A] = A \times (A \to \mathrm{comm}).$$

Assignment and lookup

- The type for the assignment constant is now

$$(A \times (A \to \text{comm})) \to A \to \text{comm}$$

- Now it's easy to interpret, using projection.
- Lookup is similarly simple: just use the other projection

$$(A \times (A \to \text{comm})) \to A$$

A cell strategy?

- How can we implement a cell strategy of type

(var[A] → comm) → comm ?

r

r

⋮

s

⋮

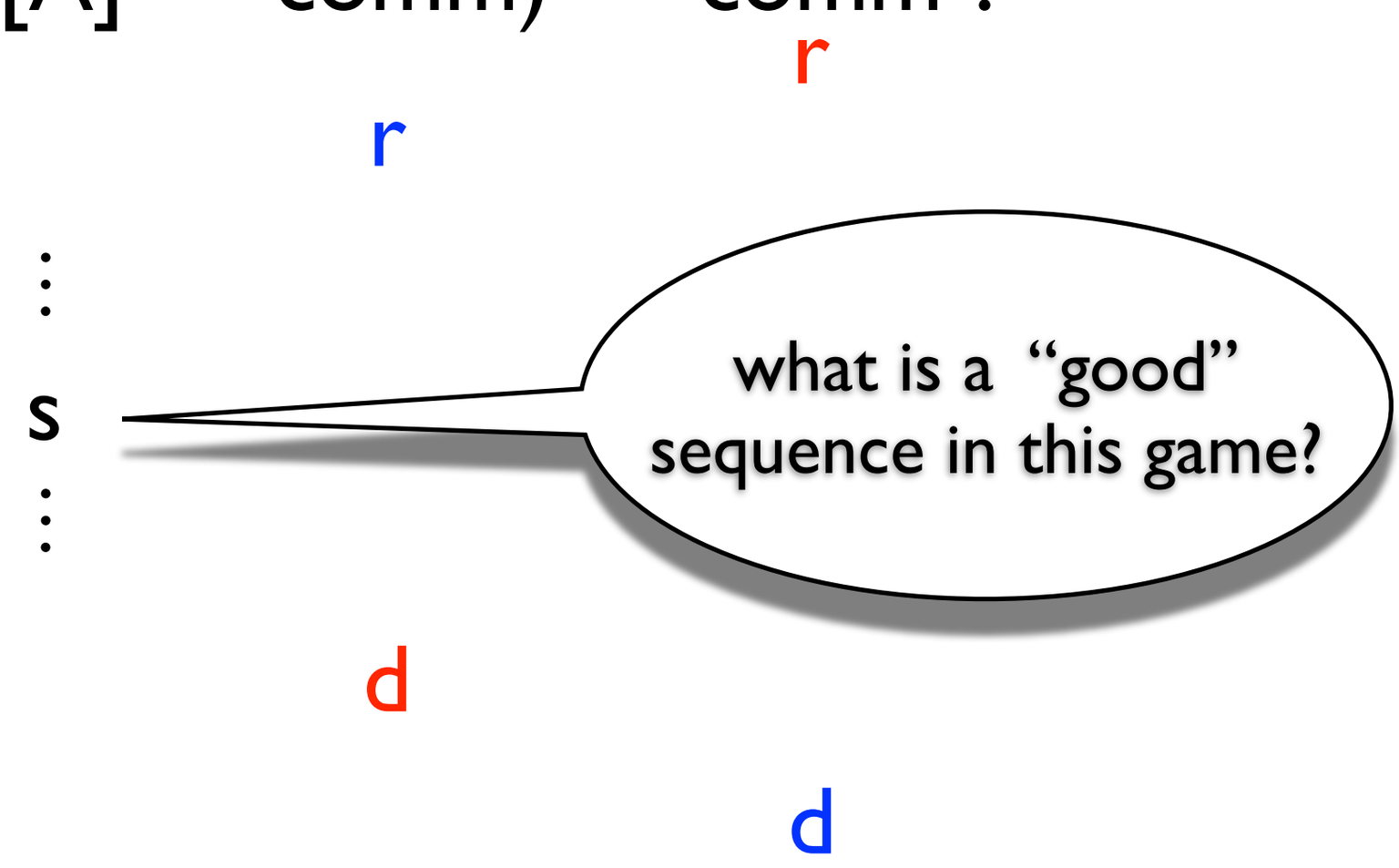
d

d

A cell strategy?

- How can we implement a cell strategy of type

$$(\mathrm{var}[A] \to \mathrm{comm}) \to \mathrm{comm}\ ?$$



cells

$$(A \times (A \to \mathrm{comm}) \to \mathrm{comm}) \to \mathrm{comm}$$

cells

$$(A \times (A \to \mathrm{comm}) \to \mathrm{comm}) \to \mathrm{comm}$$

r

cells

$$(A \times (A \to \mathrm{comm}) \to \mathrm{comm}) \to \mathrm{comm}$$

r

r

cells

$$(A \times (A \to \mathrm{comm}) \to \mathrm{comm}) \to \mathrm{comm}$$

r

r

r

cells

$$(A \times (A \to \mathrm{comm}) \to \mathrm{comm}) \to \mathrm{comm}$$

r

r

r

d

cells

$$(A \times (A \to comm) \to comm) \to comm$$

r

r

r

d

r

cells

$$(A \times (A \to comm) \to comm) \to comm$$

r

r

r

d

r

d

cells

$$(A \times (A \to comm) \to comm) \to comm$$

r

r

r

d

r

d

r

cells

$$(A \times (A \to comm) \to comm) \to comm$$

r

r

r

d

r

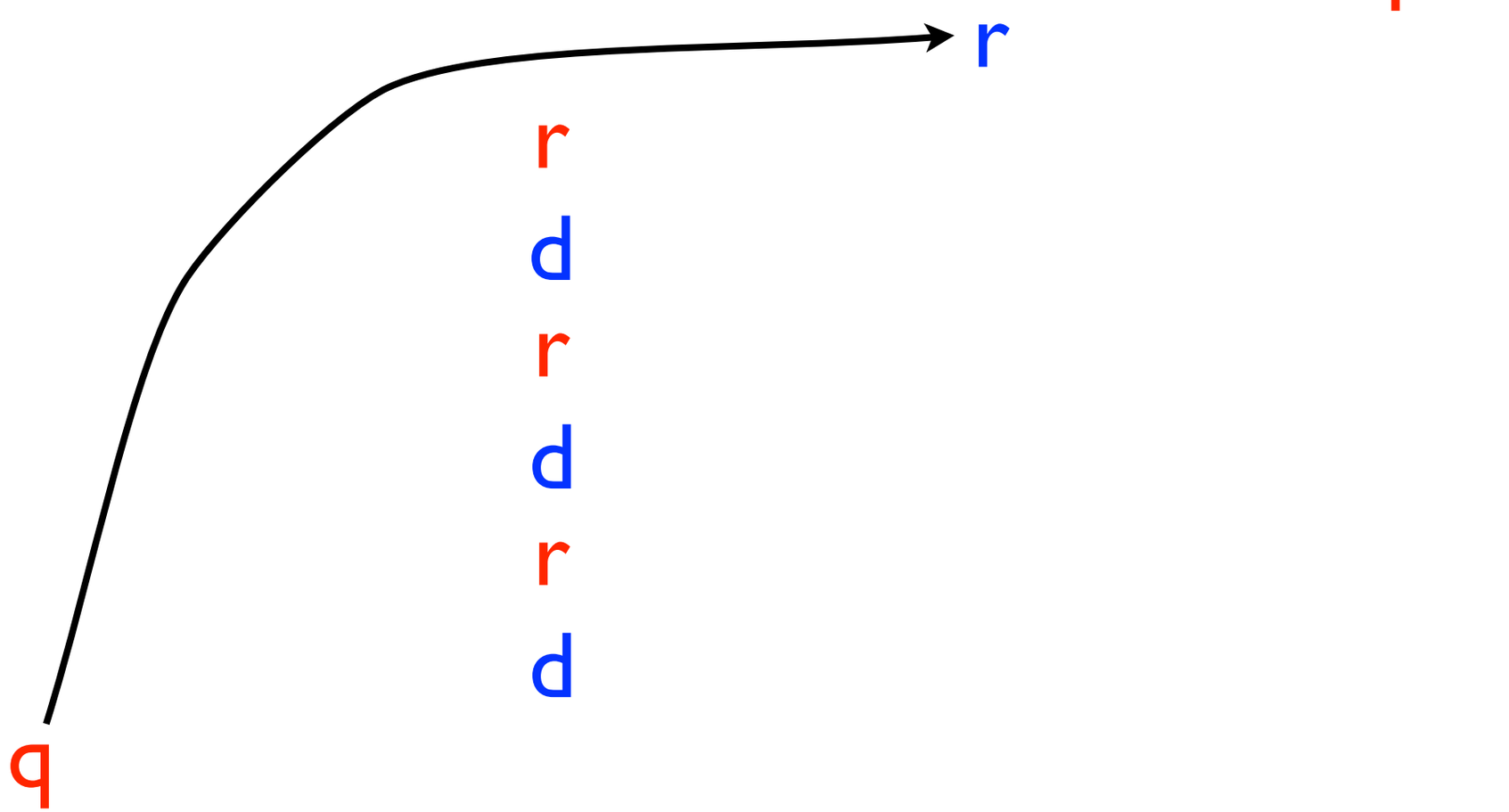
d

r

d

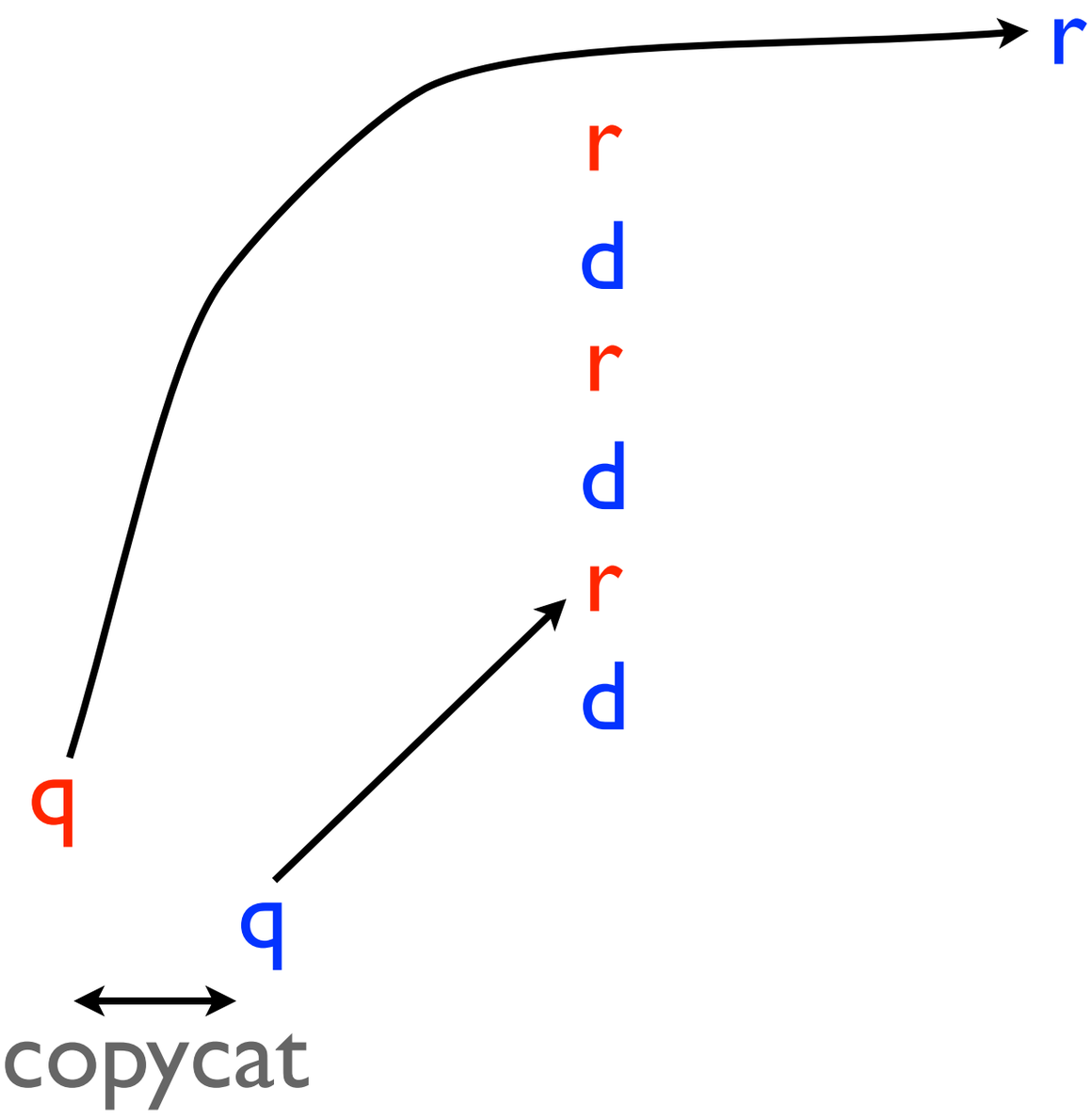
cells

$$(A \times (A \to comm) \to comm) \to comm$$



cells

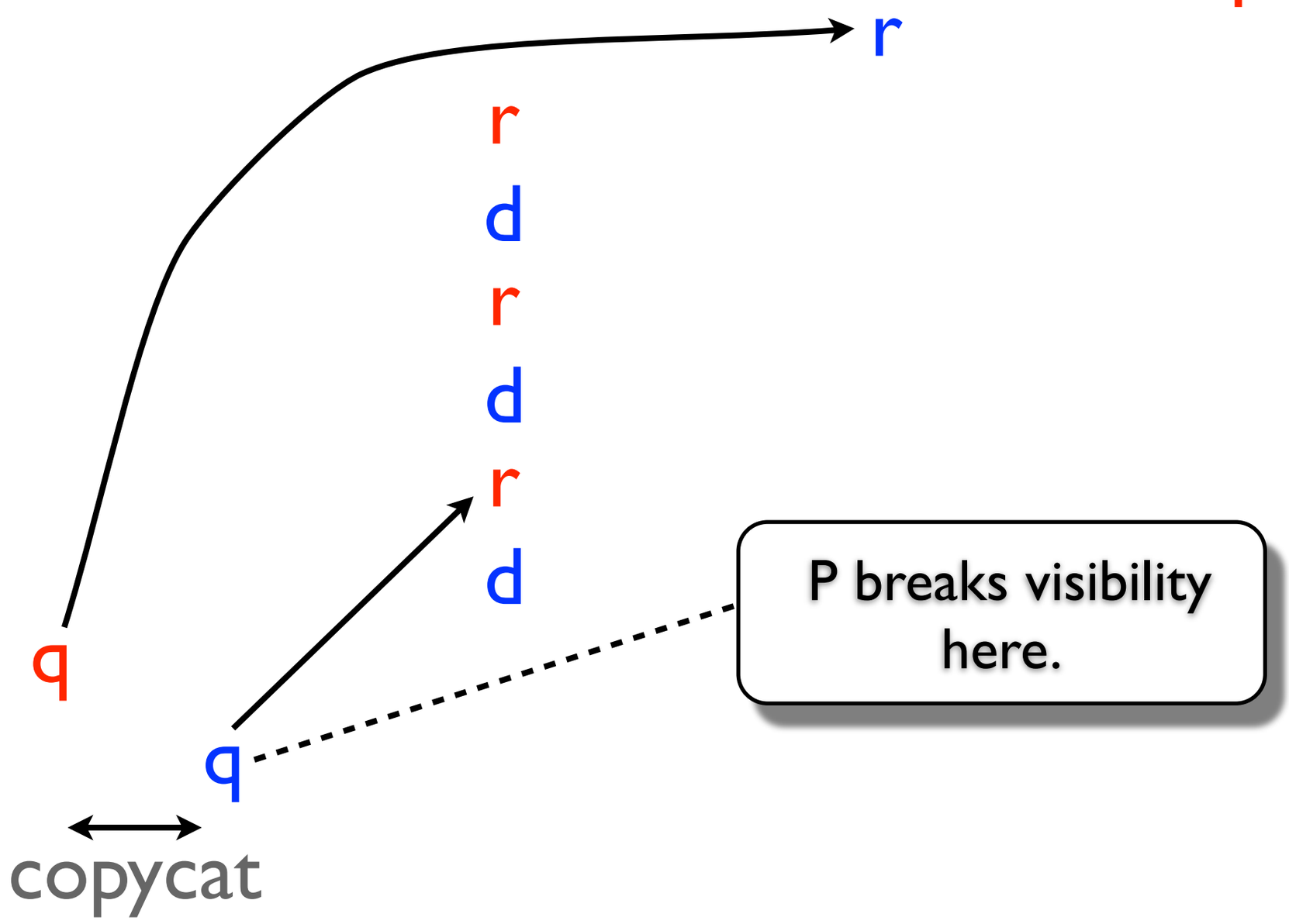
$$(A \times (A \to \mathrm{comm}) \to \mathrm{comm}) \to \mathrm{comm}$$



cells

$$(A \times (A \to comm) \to comm) \to comm$$

r



Factorization again

Theorem (approx)
[Abramsky, Honda,M 1998]
Every finite well-bracketed strategy on a type $A \to$ comm is of the form σ ; cell^n for some *P-visible*
$\sigma : A \to (\mathrm{var}[\mathrm{comm}]^n \to \mathrm{comm})$.

Factorization again

Theorem (approx)
[Abramsky, Honda,M 1998]
Every finite well-bracketed strategy on a type A → comm is of the form σ ; cell^n for some *P-visible* σ :A → (var[comp

Proof: turn all the violations of P-visibility into violations of O-visibility performed by the cell.

Factorization again

Theorem (approx)
[Abramsky, Honda,M 1998]
Every finite well-bracketed strategy on a type $A \to$ comm is of the form σ ; cell^n for some *P-visible*
$\sigma : A \to (\mathrm{var}[\mathrm{comm}]^n \to \mathrm{comm})$.

Factorization again

Theorem (approx)
[Abramsky, Honda,M 1998]
Every finite well-bracketed strategy on a type $A \to$ comm is of the form σ ; $cell^n$ for some *P-visible*
$\sigma : A \to (var[comm]^n \to comm)$.

This means that *arbitrary* well-bracketed strategies can be expressed as a composition of cells storing comm and boolean types.

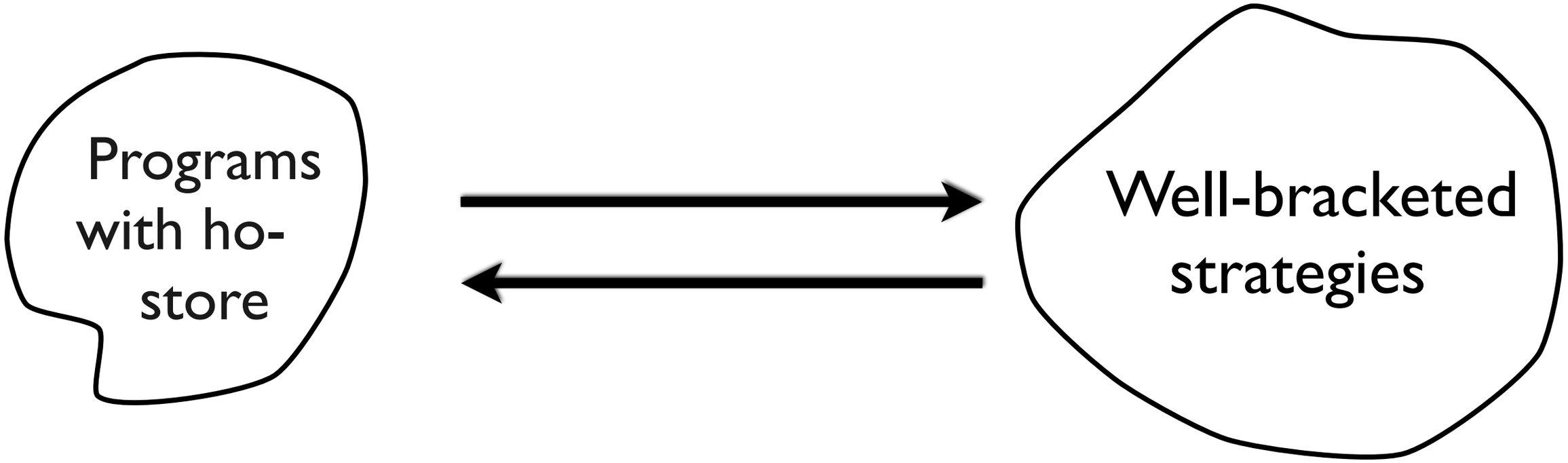
Definability again

- Our cell strategies allow us to interpret a language where terms of any type can be stored in the variables.
- The factorization result means that we have a definability result once more: every finite strategy is the denotation of some term in this language.

Literature note 3:

- Laird has shown how the ideas at play here can be expressed algebraically, in terms of his *sequoidal categories*.
- Laird, *A Categorical Semantics of Higher-Order Store*, CTCS 2002.

Definability for Higher-Order Store



Closing remarks

Conditions classify programs

λ-calculus	⟷	finite, total, innocent
PCF	⟷	innocent, P-bracketed
PCF+callcc	⟷	innocent
Idealised Algol	⟷	P-bracketed, P-visible
Higher-type Store	⟷	P-bracketed

O is unconstrained

- Our behavioural constraints have all been expressed as conditions on *strategies*.
- O is free to behave as he wishes.
- This means it makes sense to allow, e.g., a PCF term to interact in an IA context.

Full abstraction

- In each case, we can lift our *full completeness* (definability) result to a *full abstraction* result by means of a quotient:

$$\sigma \approx \tau : A \text{ if and only if for all } \alpha : A \to \mathrm{comm},$$
$$\sigma;\alpha = \tau;\alpha.$$

In the case of the imperative languages, this quotient can be defined directly.

Fin