Disciplining Orchestration and Conversation in Service-Oriented Computing

Ivan Lanese (Bologna), **Vasco T. Vasconcelos** (Lisbon), Francisco Martins (Lisbon), Antonio Ravara (Lisbon)

The problem: change

The problem: change

- Ubiquitous in business:

The problem: change

- Ubiquitous in business:

  New technologies, acquisitions, mergers.

The problem: change

- Ubiquitous in business:

  New technologies, acquisitions, mergers.

- Evil to programmers:

The problem: change

- Ubiquitous in business:

  New technologies, acquisitions, mergers.

- Evil to programmers:

  Separation of soft development and soft maintenance is vanishing.

Existing technologies won’t do

Existing technologies won’t do

- **Objects** incapable to cope with the rapidly change of software systems

Existing technologies won't do

- **Objects** incapable to cope with the rapidly change of software systems
- **Components** are usually delivered physically; do not take advantage of internet-based computing

Accommodating change: software services

- Definitions abound. Here's a recent one:

> A coarse grain, discoverable entity that [..] interacts with applications and other services.
>
> Elfatatry, CACM, Aug 2007

Aim

- Develop formal bases for Service Oriented Computing (SOC):
  - including models and techniques
  - allowing for safe development of applications
  - check that systems provide the required functionalities

What this talk in **not** about

What this talk in **not** about

- Web services

What this talk in **not** about

- Web services
- XML, SOAP, WSDL, ...

What this talk in **not** about

- Web services
- XML, SOAP, WSDL, ...
- Service discovery, negotiation, brokerage

Outline

- A motivating example
- Semantics
- Analyses
- Conclusion

Example: booking an hotel

- A process

(date) {query-the-hotel-db}.price

Example: booking an hotel

- A process

receive a value

(date) {query-the-hotel-db}.price

Example: booking an hotel

- A process

receive a value

some computation

(date) {query-the-hotel-db}.price

Example: booking an hotel

- A process

receive a value

some computation

send a value

(date) {query-the-hotel-db}.price

Example: booking an hotel

- A process

receive a value

some computation

send a value

(date) {query-the-hotel-db}.price

- A service

Example: booking an hotel

- A process

receive a value

some computation

send a value

(date) {query-the-hotel-db}.price

- A service

bologna => (date) {query-the-hotel-db}.price

Example: booking an hotel

- A process

receive a value

some computation

send a value

(date) {query-the-hotel-db}.price

- A service

bologna => (date) {query-the-hotel-db}.price

service name

Example: booking an hotel

- A process

receive a value

some computation

send a value

(date) {query-the-hotel-db}.price

- A service

bologna => (date) {query-the-hotel-db}.price

service name

right arrow indicates provider

Example: the client

- A service consumer

bologna <= 31Jul2007.(price) {use-price}

Example: the client

- A service consumer

    bologna <= 31Jul2007.(price) {use-price}

- An interaction

Example: the client

- A service consumer

```
bologna <= 31Jul2007.(price) {use-price}
```

- An interaction

```
bologna => ...  |  bologna <= ...
```

Example: the client

- A service consumer

bologna <= 31Jul2007.(price) {use-price}

- An interaction

bologna => ... | bologna <= ...

provider

Example: the client

- A service consumer

bologna <= 31Jul2007.(price) {use-price}

- An interaction

bologna => ... | bologna <= ...

provider consumer

Example: the client

- A service consumer

bologna <= 31Jul2007.(price) {use-price}

- An interaction

parallel composition

bologna => ... | bologna <= ...

provider

consumer

Example: a broker comes and...

Example: a broker comes and...

...calls three services

Example: a broker comes and...

...calls three services

bologna <= date.(price1) ... |
azores <= date.(price2) ... |
lisbon <= date.(price3) ...

Example: a broker comes and...

...calls three services

```
bologna <= date.(price1) ... |
azores <= date.(price2) ... |
lisbon <= date.(price3) ...
```

- How to collect the three prices in a single process, for further processing?

Streams to the rescue

- A service orchestrator

```
stream
  bologna <= date.(price1).feed price1  |
  azores <= date.(price2).feed price2  |
  lisbon <= date.(price3).feed price3
as f in
  f(x).f(y).{publish-the-min-of-x-and-y}
```

Streams to the rescue

- A service orchestrator

write into the stream

```
stream
  bologna <= date.(price1).feed price1 |
  azores <= date.(price2).feed price2 |
  lisbon <= date.(price3).feed price3
as f in
  f(x).f(y).{publish-the-min-of-x-and-y}
```

Streams to the rescue

- A service orchestrator

```
stream
  bologna <= date.(price1).feed price1 |
  azores <= date.(price2).feed price2 |
  lisbon <= date.(price3).feed price3
as f in
  f(x).f(y).{publish-the-min-of-x-and-y}
```

write into the stream

read from the stream

Common patterns deserve abbreviations

```
(call bologna(date) |
 call azores(date) |
 call lisbon(date)) > x y >
  {publish-the-min-of-x-and-y}
```

Common patterns deserve abbreviations

call service
bologna; write the result
into the pipe

```
(call bologna(date) |
 call azores(date) |
 call lisbon(date)) > x y >
  {publish-the-min-of-x-and-y}
```

Common patterns deserve abbreviations

call service bologna; write the result into the pipe

read two values from the pipe; call them x and y

```
(call bologna(date) |
 call azores(date) |
 call lisbon(date)) > x y >
  {publish-the-min-of-x-and-y}
```

Common patterns deserve abbreviations

call service bologna; write the result into the pipe

(**call** bologna(date) |
call azores(date) |
call lisbon(date)) **>** x y **>**
{publish-the-min-of-x-and-y}

read two values from the pipe; call them x and y

Inspired in Orc!

Example: service composition

```
broker => (date).(
  (call bologna(date) |
   call azores(date) |
   call lisbon(date)) > x y >
   call min(x,y) > m > m)
```

Example: service composition

a service definition

```
broker => (date).(
  (call bologna(date) |
   call azores(date) |
   call lisbon(date)) > x y >
   call min(x,y) > m > m)
```

Example: service composition

a service definition

```
broker => (date).(
  (call bologna(date) |
   call azores(date) |
   call lisbon(date)) > x y >
   call min(x,y) > m > m)
```

call a service to compute the min

Example: service composition

a service definition

```
broker => (date).(
  (call bologna(date) |
   call azores(date) |
   call lisbon(date)) > x y >
   call min(x,y) > m > m)
```

call a service to compute the min

read the result

Example: service composition

a service definition

```
broker => (date).(
  (call bologna(date) |
   call azores(date) |
   call lisbon(date)) > x y
   call min(x,y) > m > m)
```

return it

call a service to compute the min

read the result

Clients won’t notice the difference

- The client

broker <= 31Jul2007.(price) {use-price}

- Interaction as before

broker <= ... | broker => ...

Clients won’t notice the difference

- The client

```
broker <= 31Jul2007.(price) {use-price}
```

- Interaction as before

```
broker <= ...  |  broker => ...
```

Central to services!

Syntax

P, Q	$::=$		*Processes*
		$P \mid Q$	Parallel composition
	$\mid$	$(\nu a)P$	Name restriction
	$\mid$	$\mathbf{0}$	Terminated process
	$\mid$	X	Process variable
	$\mid$	$\mathsf{rec}\, X.P$	Recursive process definition
	$\mid$	$a \Rightarrow P$	Service definition
	$\mid$	$a \Leftarrow P$	Service invocation
	$\mid$	$v.P$	Value sending
	$\mid$	$(x)P$	Value reception
	$\mid$	$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q$	Stream
	$\mid$	$\mathsf{feed}\ v.P$	Feed the process' stream
	$\mid$	$f(x).P$	Read from a stream

Process calculus

Service

Protocol

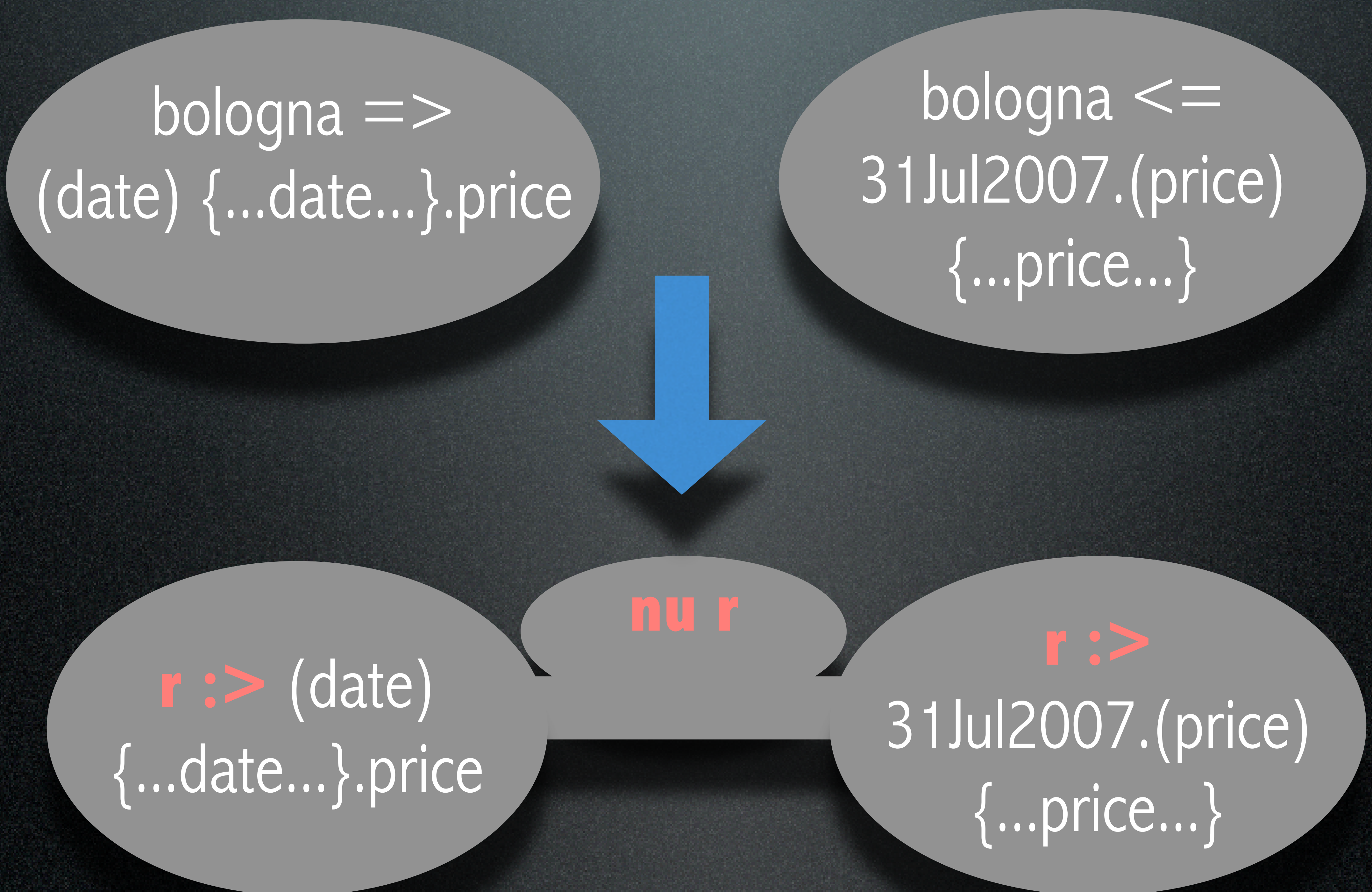
Stream

Operational semantics: service invocation

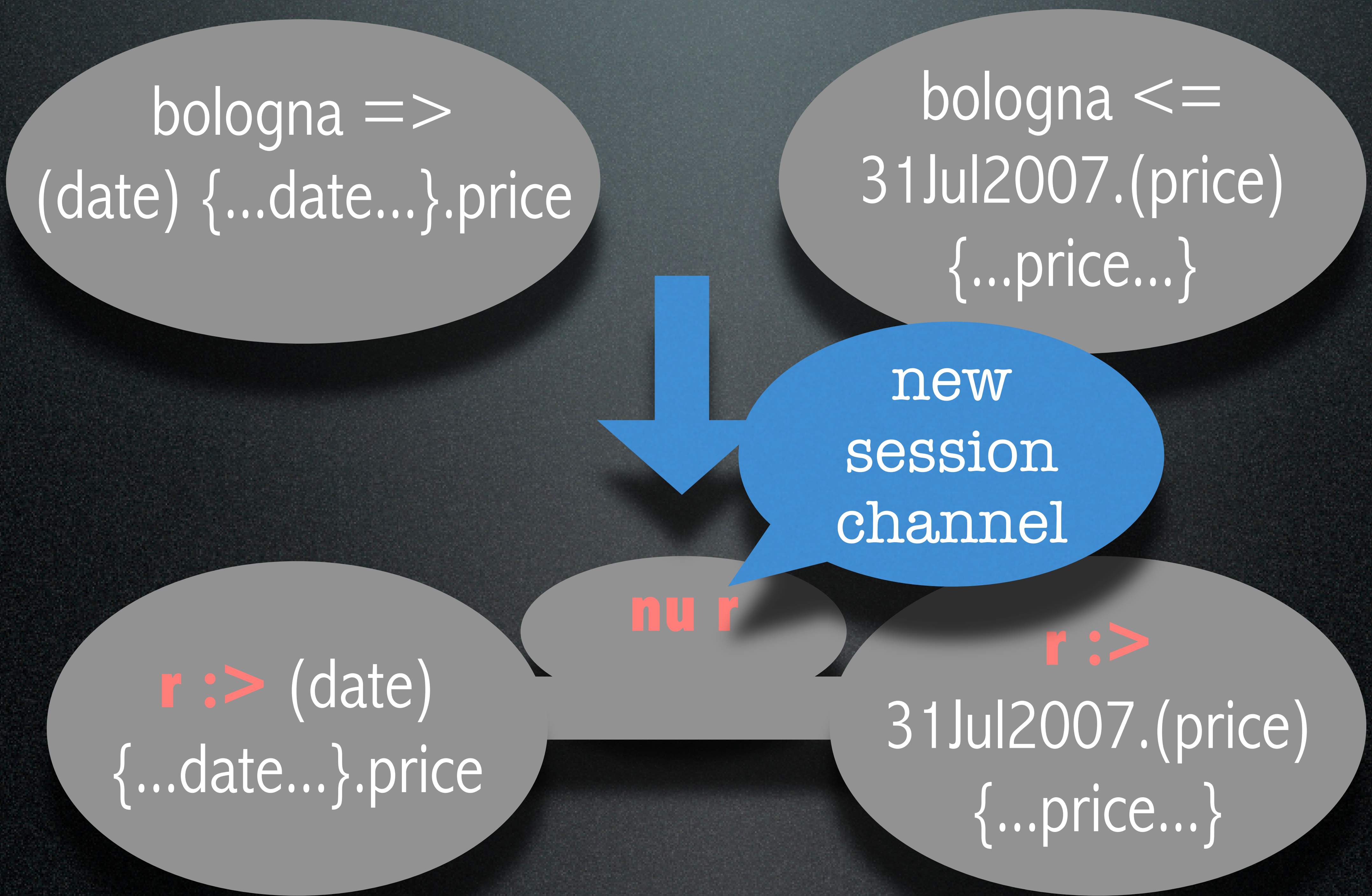
bologna =>
(date) {...date...}.price

bologna <=
31Jul2007.(price)
{...price...}

Operational semantics: service invocation

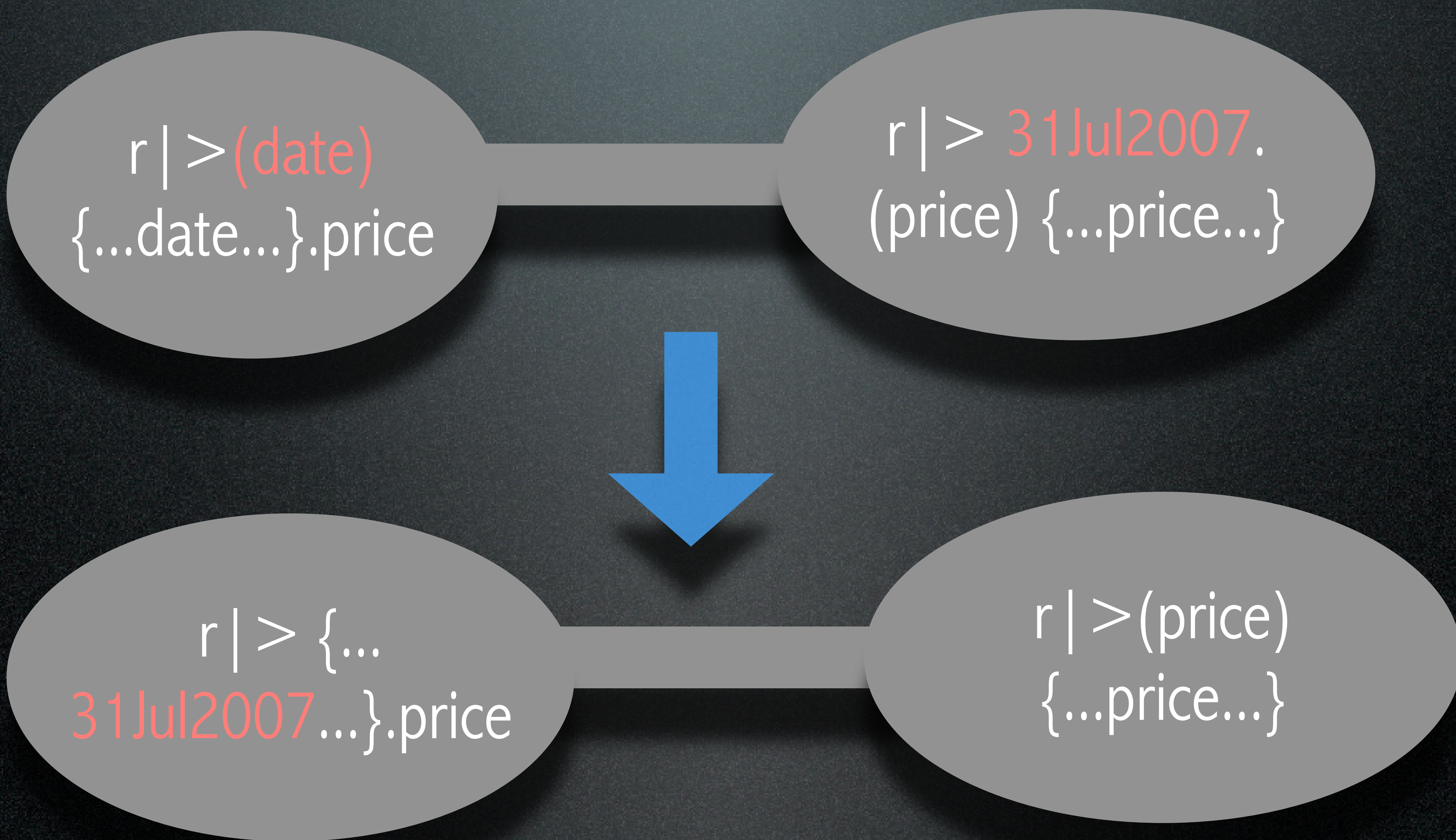


Operational semantics: service invocation



Operational semantics: protocol

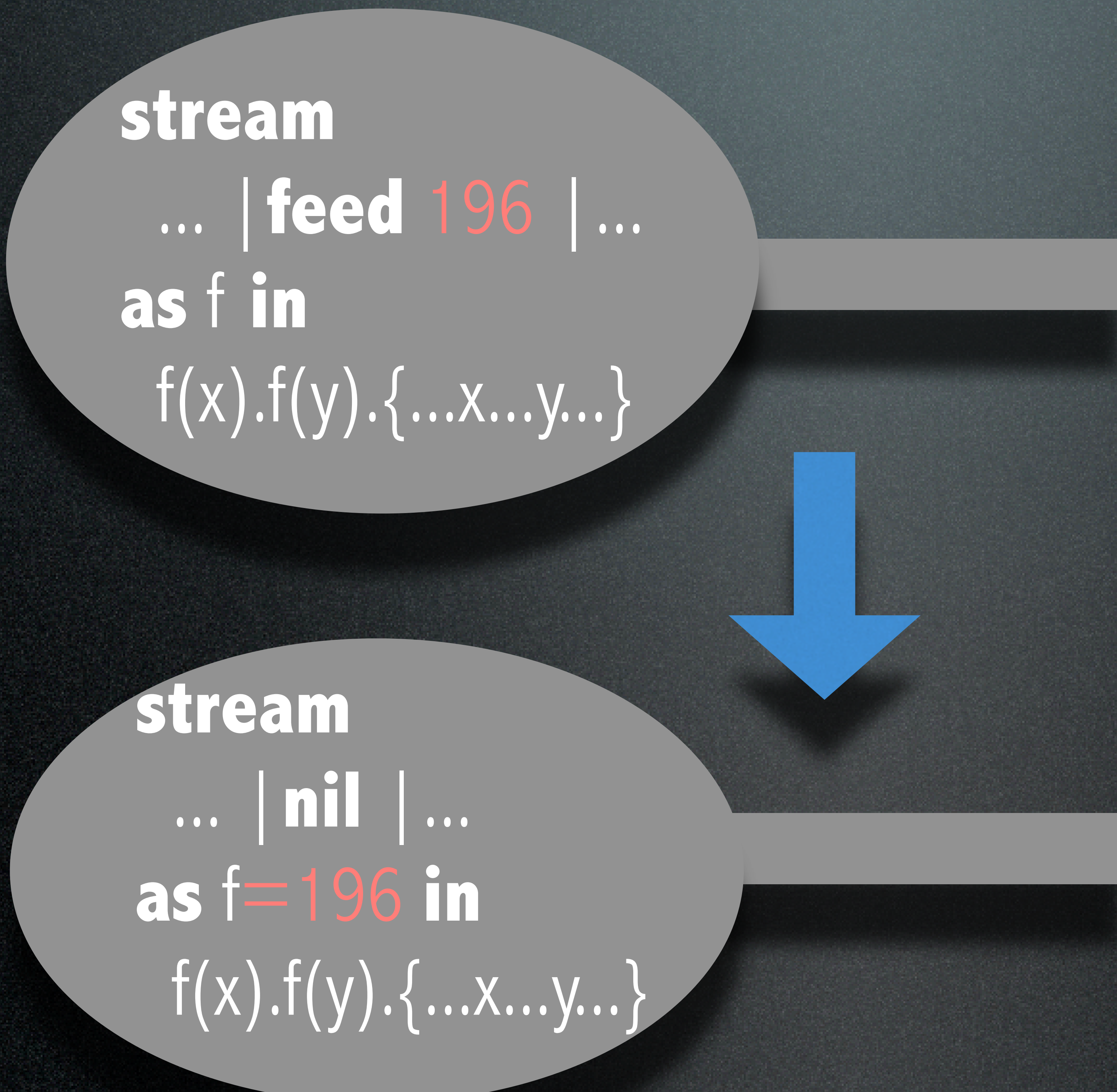
Operational semantics: protocol



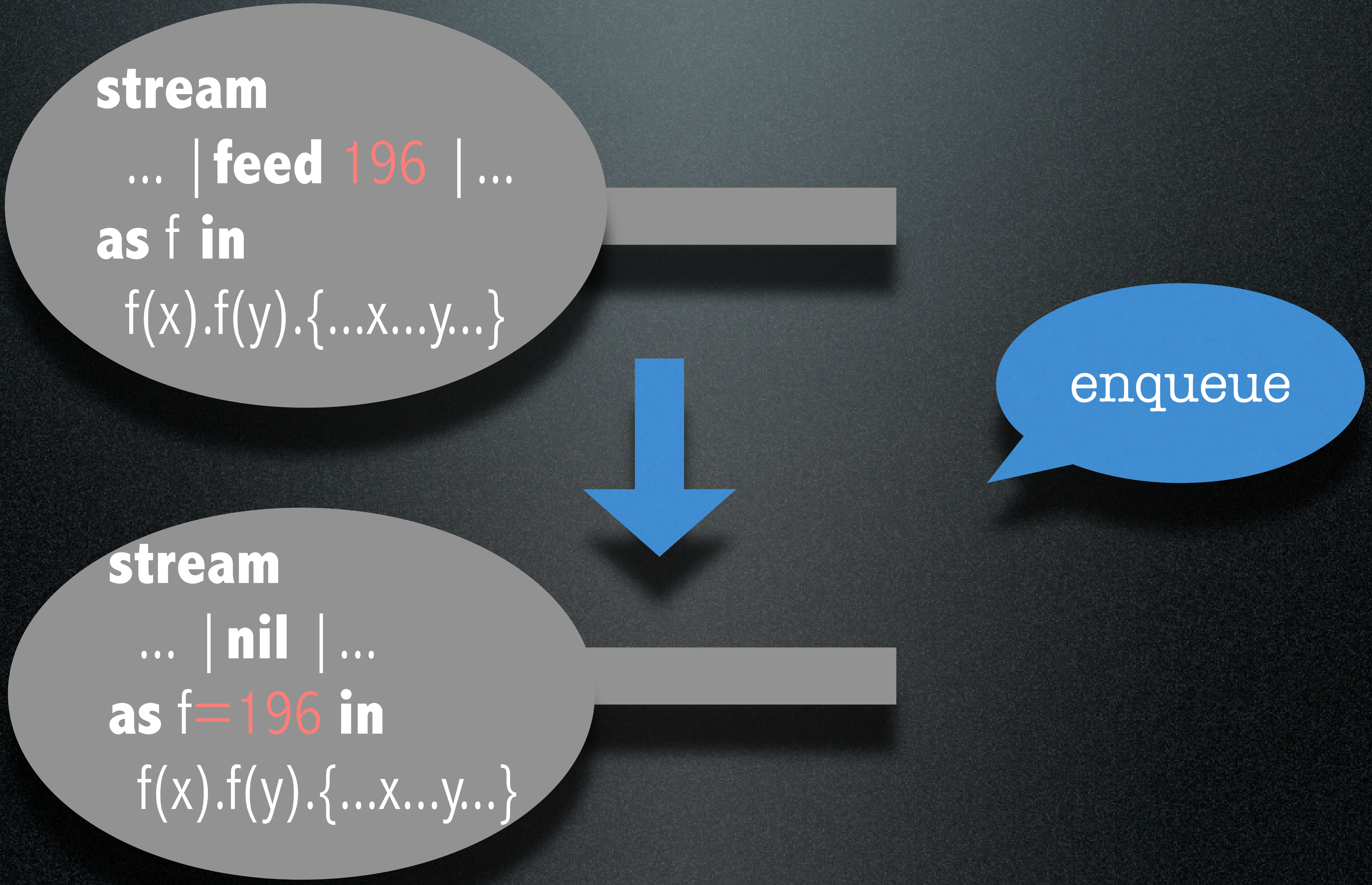
Operational semantics: streams

stream
… |**feed** 196 |…
as f **in**
f(x).f(y).{…x…y…}

Operational semantics: streams



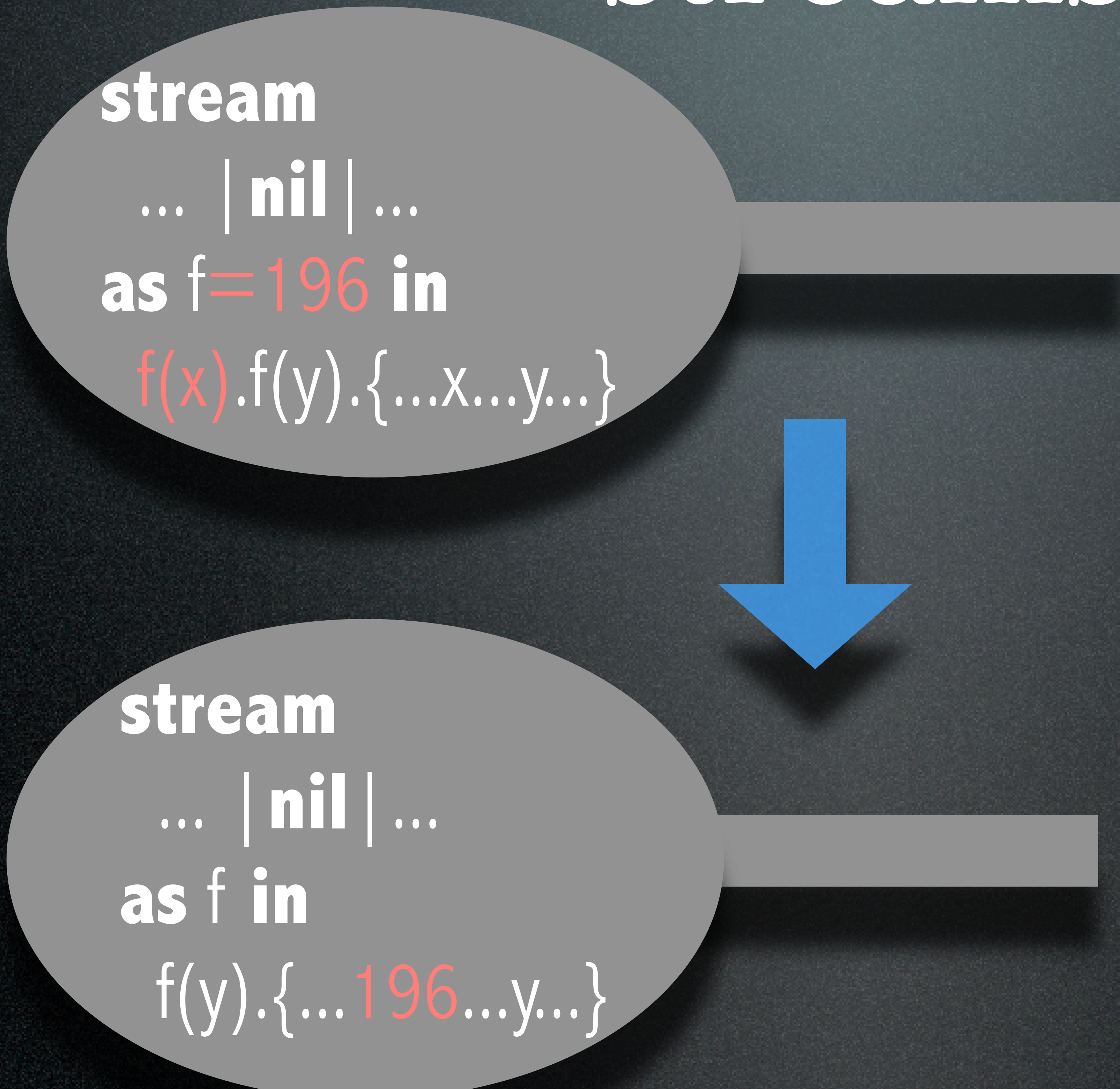
Operational semantics: streams



Operational semantics: streams

stream
 … | **nil** | …
as f=196 **in**
 f(x).f(y).{…x…y…}

Operational semantics: streams



Operational semantics: streams

stream
… | **nil** | …
as f=196 **in**
f(x).f(y).{…x…y…}

dequeue

stream
… | **nil** | …
as f **in**
f(y).{…196…y…}

Reduction semantics

- Structural congruence - allows the syntactic rearrangement of terms

$$(\nu n)P|Q \quad \equiv \quad (\nu n)(P|Q) \quad \text{if } n \notin \text{fn}(Q)$$

Reduction semantics

- Structural congruence - allows the syntactic rearrangement of terms

$$(\nu n)P|Q \equiv (\nu n)(P|Q) \quad \text{if } n \notin \text{fn}(Q)$$

- Allows reduction at certain places in a term

$$\frac{\mathcal{C}[\![]\!] \text{ does not bind } w \text{ or } f}{\mathsf{stream}\ P\ \mathsf{as}\ f = \vec{v} :: w\ \mathsf{in}\ \mathcal{C}[\![f(x).Q]\!] \rightarrow \mathsf{stream}\ P\ \mathsf{as}\ f = \vec{v}\ \mathsf{in}\ \mathcal{C}[\![Q[{}^{w}/_{x}]]\!]}$$

Reduction semantics

- Structural congruence - allows the syntactic rearrangement of terms

$$(\nu n)P|Q \quad \equiv \quad (\nu n)(P|Q) \quad \text{if } n \notin \text{fn}(Q)$$

Sample rules!

- Allows reduction at certain places in a term

$$\frac{\mathcal{C}[\![]\!] \text{ does not bind } w \text{ or } f}{\textsf{stream}\ P\ \textsf{as}\ f = \vec{v} :: w\ \textsf{in}\ \mathcal{C}[\![f(x).Q]\!] \rightarrow \textsf{stream}\ P\ \textsf{as}\ f = \vec{v}\ \textsf{in}\ \mathcal{C}[\![Q[{}^{w}/_{x}]]\!]}$$

Labeled transition system

- Sample rule:

read v from stream f

$$\frac{Q \xrightarrow{f \Downarrow v} Q'}{\text{stream } P \text{ as } f = \vec{w} :: v \text{ in } Q \xrightarrow{\tau} \text{stream } P \text{ as } f = \vec{w} \text{ in } Q'}$$

- Correspondence

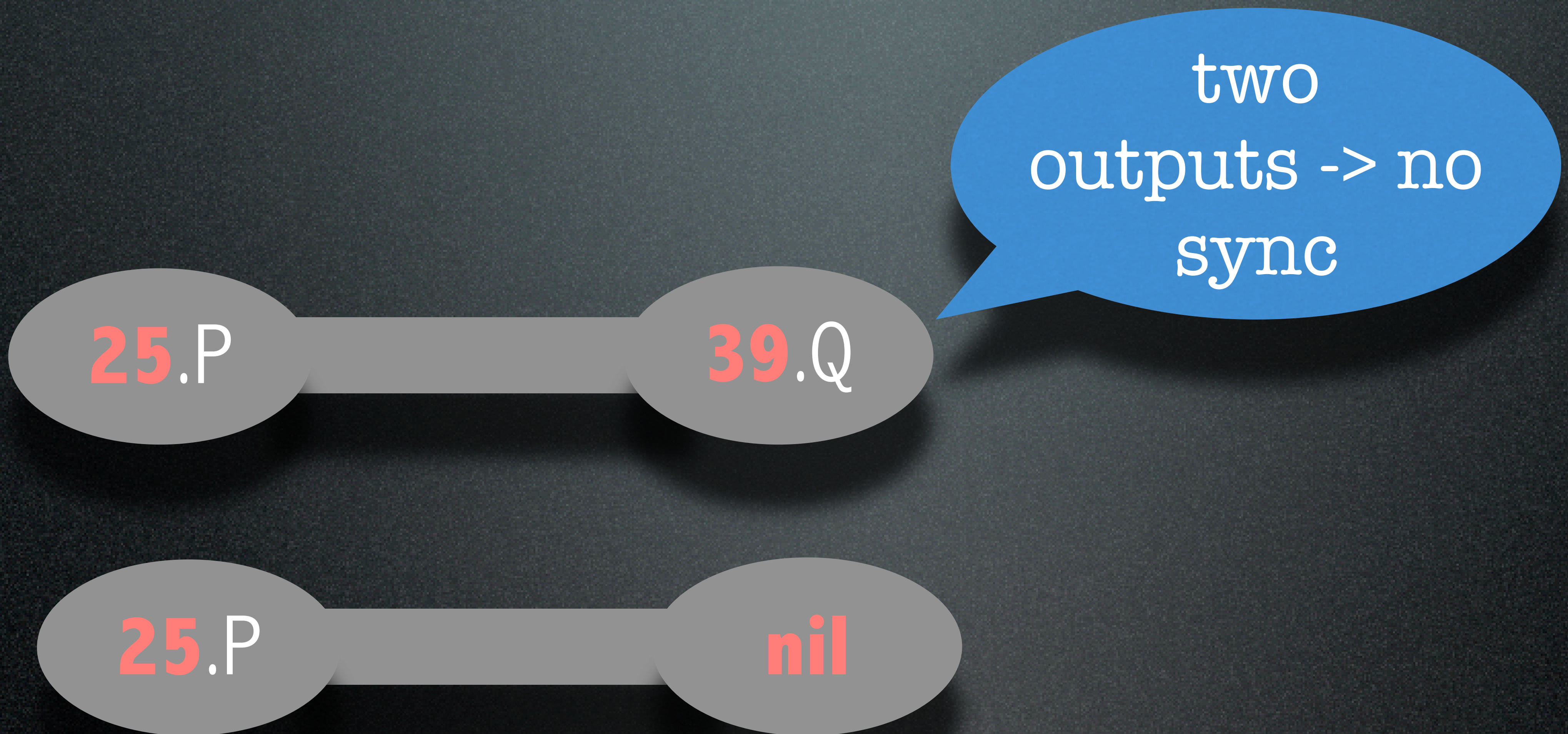
$$P \longrightarrow Q \textit{ if and only if } P \xrightarrow{\tau} Q$$

- Leads to bisimulation-based equivalences

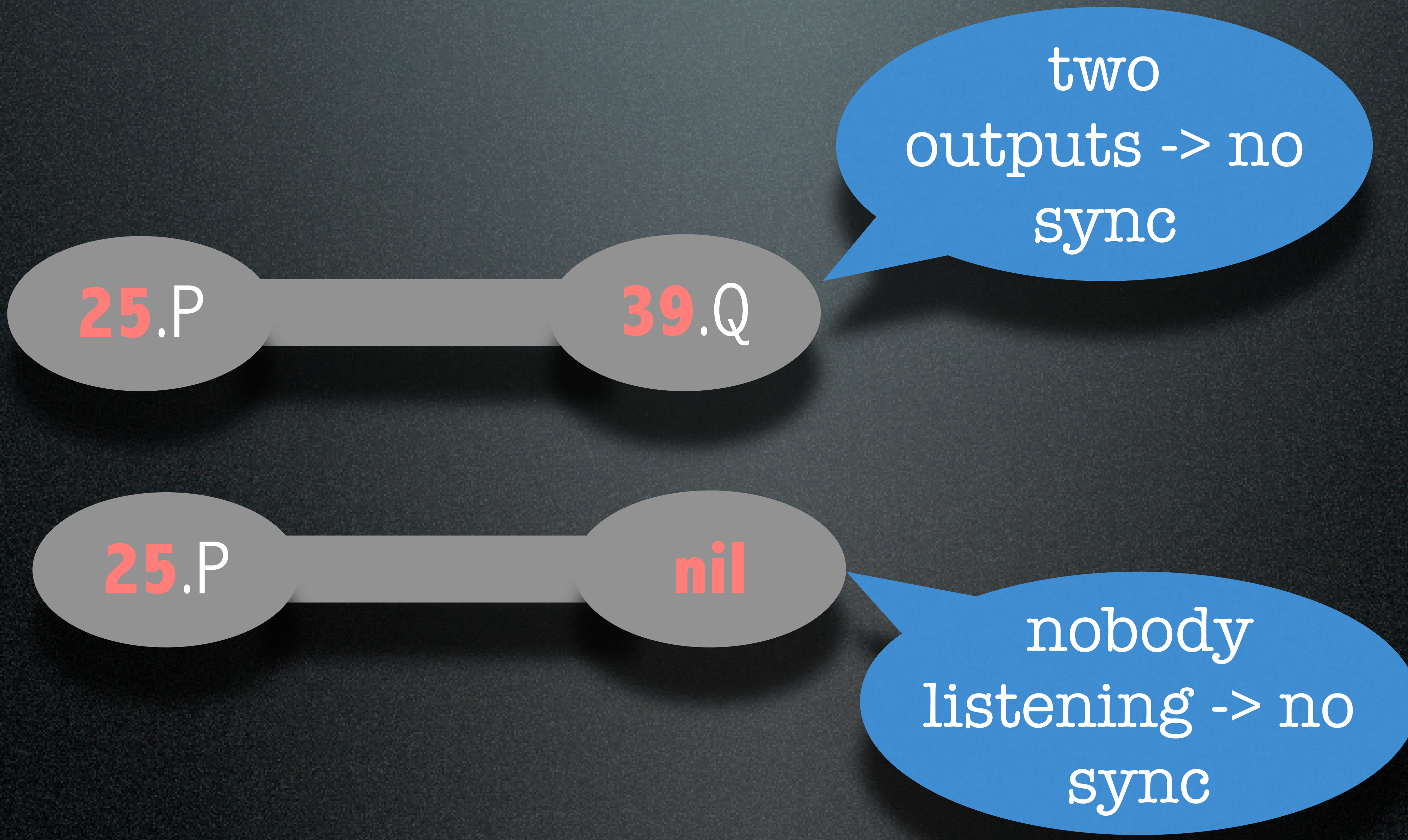
What can go wrong?
I: thread sync

What can go wrong?
I: thread sync

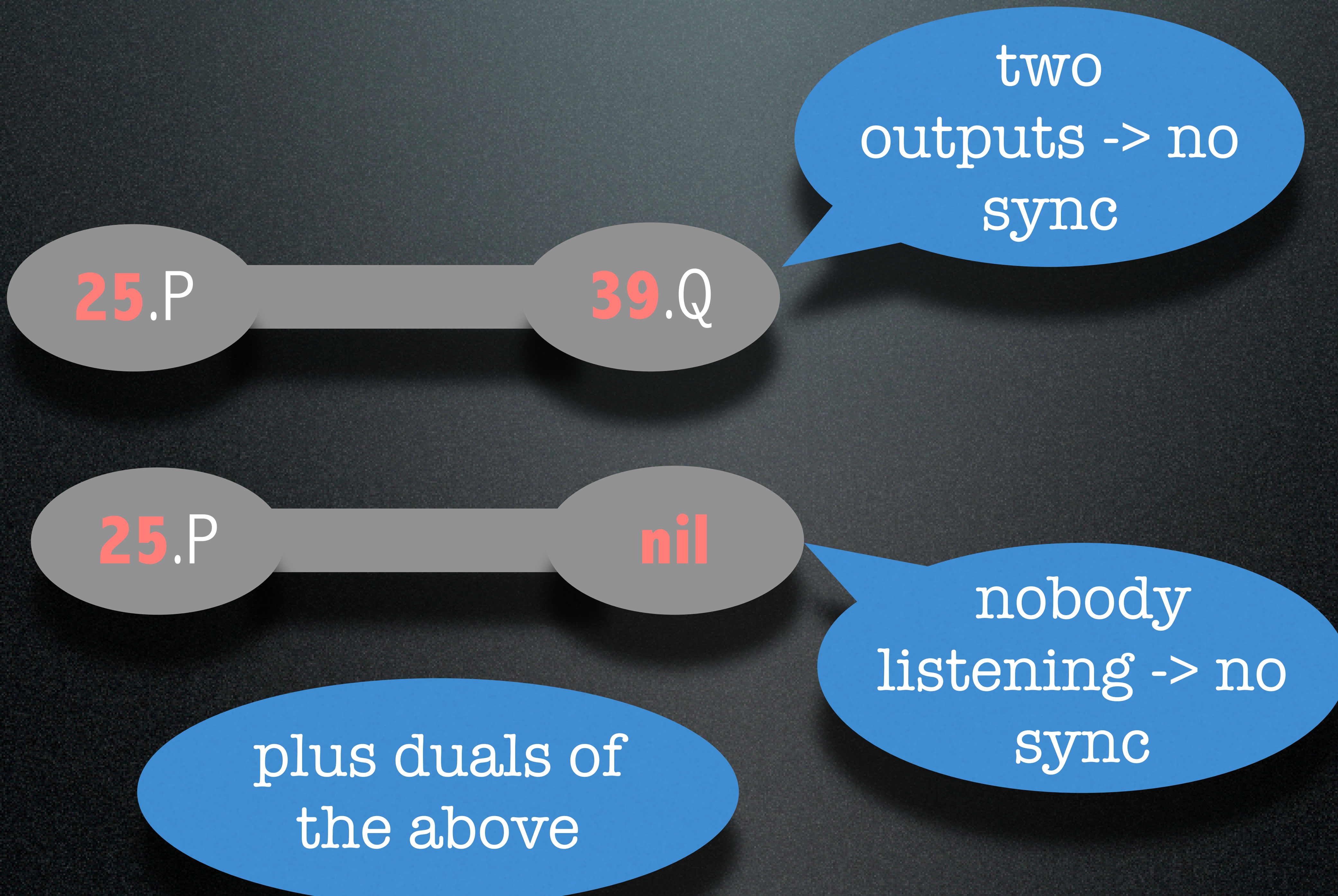
What can go wrong?
I: thread sync



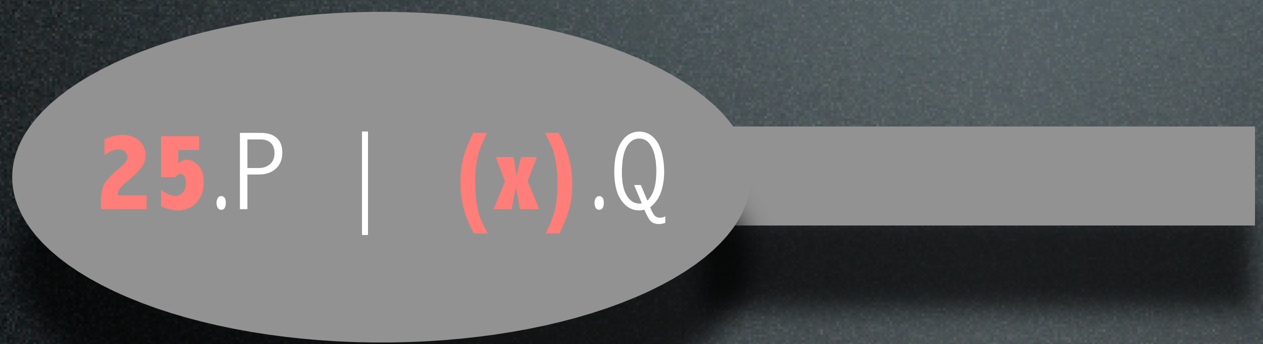
What can go wrong?
I: thread sync



What can go wrong?
I: thread sync

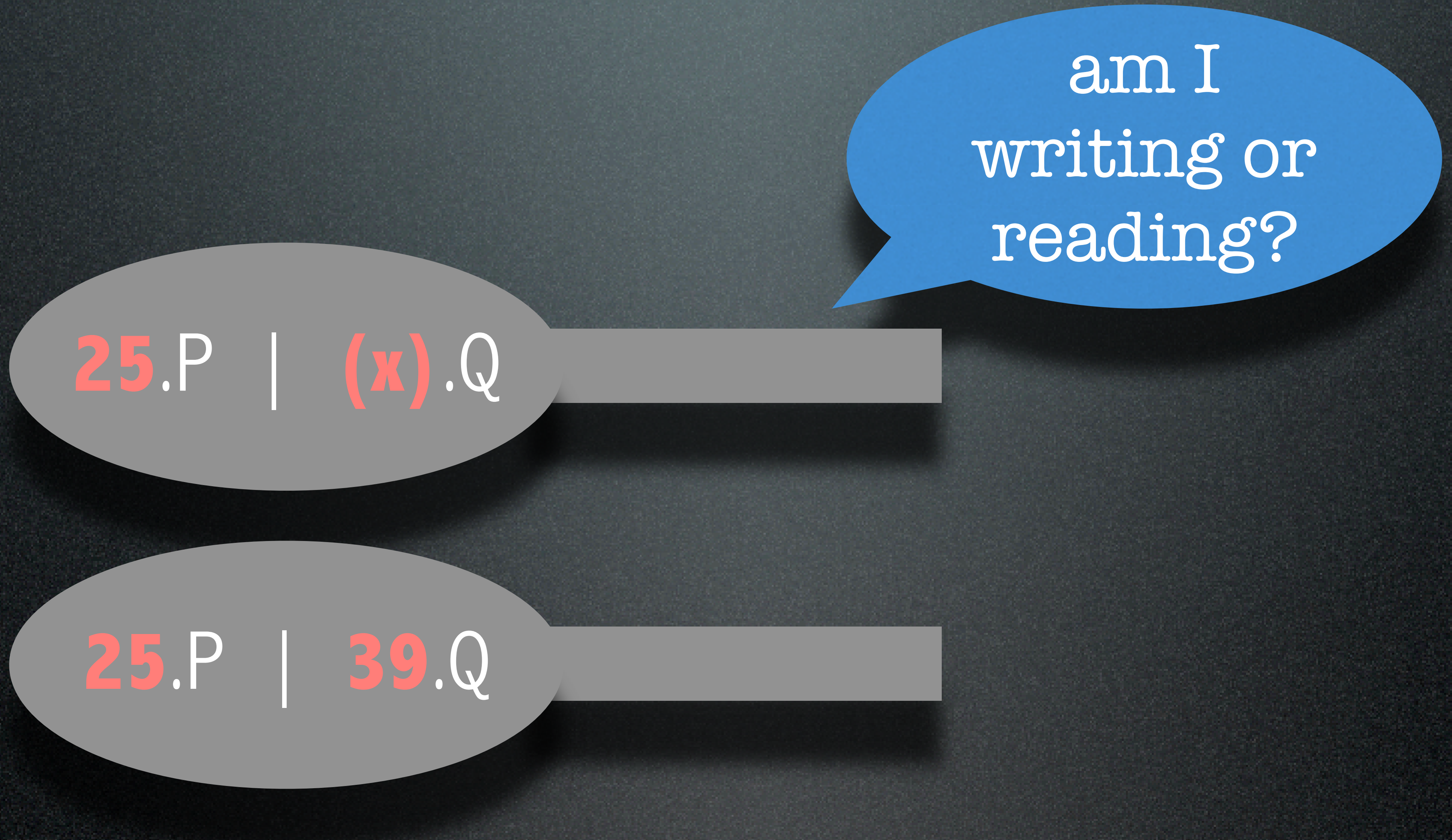


What can go wrong?
II: intra-thread comm

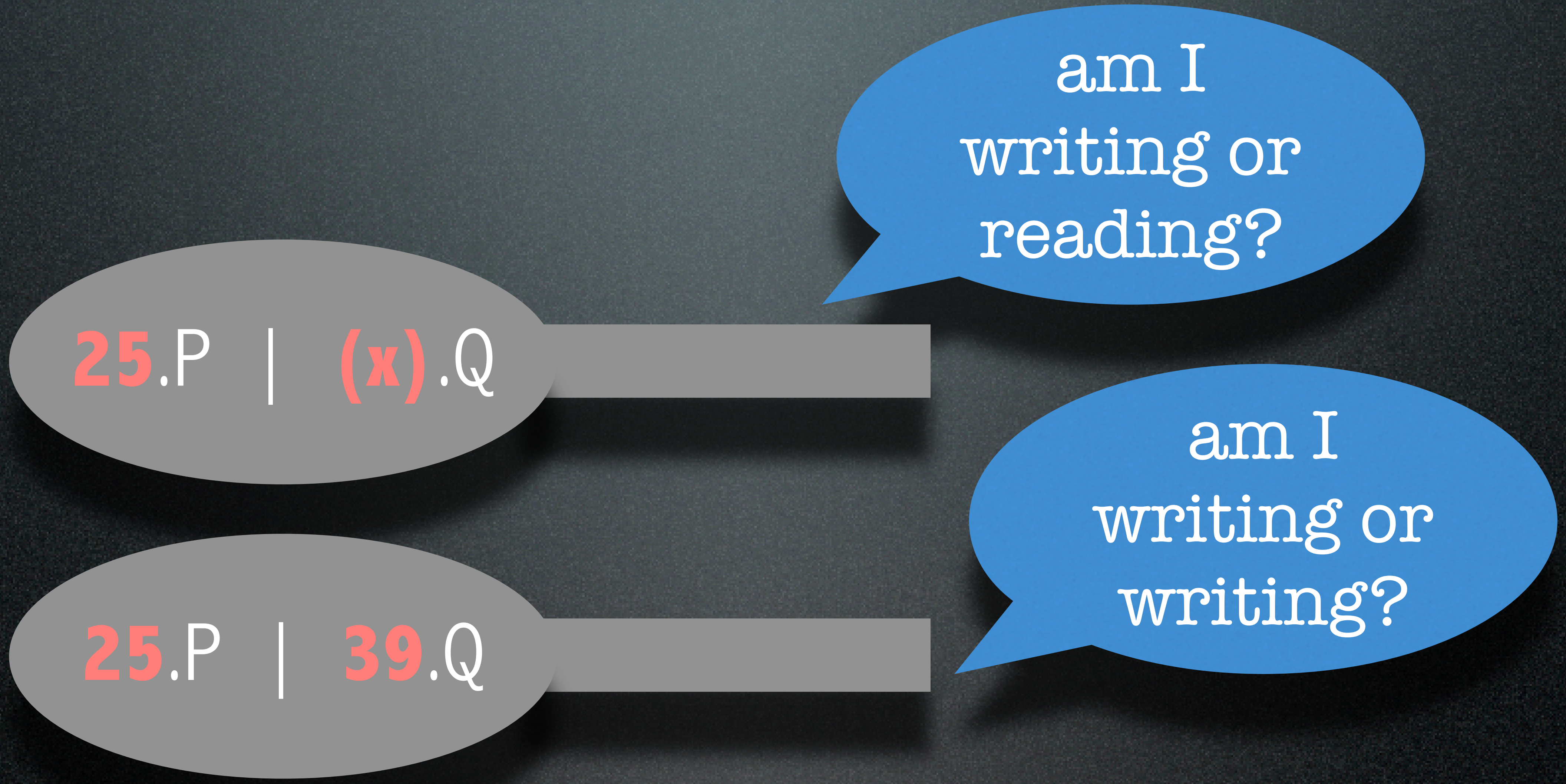


What can go wrong?
II: intra-thread comm

What can go wrong?
II: intra-thread comm



What can go wrong?
II: intra-thread comm



What can go wrong? II: intra-thread comm

am I writing or reading?

25.P | (x).Q

am I writing or writing?

25.P | 39.Q

plus duals of the above

The type of a protocol

(date) {query-the-hotel-db}.price

The type of a protocol

(date) {query-the-hotel-db}.price

?Date.!Int.**end**

The type of a protocol

(date) {query-the-hotel-db}.price

?Date.!Int.**end**

end of the protocol

The type of a protocol

no input or output here

(date) {query-the-hotel-db}.price

?Date.!Int.**end**

end of the protocol

The type of a protocol

no input or output here

(date) {query-the-hotel-db}.price

?Date.!Int.**end**

end of the protocol

31Jul2007.(price) {use-price}

The type of a protocol

no input or output here

(date) {query-the-hotel-db}.price

?Date.!Int.**end**

end of the protocol

!Date.?Int.**end**

31Jul2007.(price) {use-price}

Compatible protocols

?Date.!Int.**end**

!Date.?Int.**end**

Compatible protocols

the type
of the service
provider

?Date.!Int.**end**

!Date.?Int.**end**

Compatible protocols

the type
of the service
provider

?Date.!Int.**end**

!Date.?Int.**end**

the type of
the client

Compatible protocols

the type of the service provider

?Date.!Int.**end**

!Date.?Int.**end**

the type of the client

Compatible protocols

the type
of the service
provider

?Date.!Int.**end**

!Date.?Int.**end**

the type of
the client

Compatible
protocols ->
type safe

Types for streams

stream
 ...**feed** price1 |
 ...**feed** price2 |
 ...**feed** price3
as f **in**
 f(x).f(y).{publish-the-min-of-x-and-y}

Types for streams

stream
…**feed** price1 |
…**feed** price2 |
…**feed** price3
as f **in**
f(x).f(y).{publish-the-min-of-x-and-y}

all feeds of the same type

Types for streams

stream
...**feed** price1 |
...**feed** price2 |
...**feed** price3
as f **in**
f(x).f(y).{publish-the-min-of-x-and-y}

all feeds of the same type

all reads of the same type

Types for streams

stream
...**feed** price1 |
...**feed** price2 |
...**feed** price3
as f **in**
f(x).f(y).{publish-the-min-of-x-and-y}

all feeds of the same type

all reads of the same type

Streams are monomorphic

The type of a process is a pair

(date).
stream
  ... |...**feed** price2 |...
as f **in**
  f(x).f(y).{publish-the-min-of-x-and-y}

The type of a process is a pair

(date).
stream
  … |…**feed** price2 |…
as f **in**
  f(x).f(y).{publish-the-min-of-x-and-y}

(?Date.!Int.**end**, Int)

The type of a process is a pair

(date).
stream
  … | …**feed** price2 | …
as f **in**
  f(x).f(y).{publish-the-min-of-x-and-y}

(?Date.!Int.**end**, Int)

The type of a process is a pair

(date).
stream
... |...**feed** price2 |...
as f **in**
f(x).f(y).{publish-the-min-of-x-and-y}

(?Date.!Int.**end**, Int)

the type
of the protocol

The type of a process is a pair

(date).
stream
... | ...**feed** price2 | ...
as f **in**
f(x).f(y).{publish-the-min-of-x-and-y}

(?Date.!Int.**end**, Int)

the type of the protocol

the type of the stream

Sample rules

$$\frac{\Gamma, x\colon T' \vdash P\colon (U,T)}{\Gamma \vdash (x)P\colon (?T'.U,T)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash a\colon [U]}{\Gamma \vdash a \Rightarrow P\colon (\mathsf{end},T)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash Q\colon (\mathsf{end},T)}{\Gamma \vdash P|Q\colon (U,T)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma, f\colon \langle T\rangle \vdash Q\colon (\mathsf{end},T')}{\Gamma \vdash \mathsf{stream}\, P\, \mathsf{as}\, f\ \mathsf{in}\, Q\colon (U,T')}$$

Sample rules

input within a session

$$\frac{\Gamma, x\colon T' \vdash P\colon (U,T)}{\Gamma \vdash (x)P\colon (?T'.U,T)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash a\colon [U]}{\Gamma \vdash a \Rightarrow P\colon (\mathsf{end},T)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash Q\colon (\mathsf{end},T)}{\Gamma \vdash P|Q\colon (U,T)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma, f\colon \langle T\rangle \vdash Q\colon (\mathsf{end},T')}{\Gamma \vdash \mathsf{stream}\; P \;\mathsf{as}\; f \;\mathsf{in}\; Q\colon (U,T')}$$

Sample rules

$$\frac{\Gamma, x \colon T' \vdash P \colon (U, T)}{\Gamma \vdash (x)P \colon (?T'.U, T)}$$

$$\frac{\Gamma \vdash P \colon (U, T) \quad \Gamma \vdash a \colon [U]}{\Gamma \vdash a \Rightarrow P \colon (\mathsf{end}, T)}$$

$$\frac{\Gamma \vdash P \colon (U, T) \quad \Gamma \vdash Q \colon (\mathsf{end}, T)}{\Gamma \vdash P|Q \colon (U, T)}$$

$$\frac{\Gamma \vdash P \colon (U, T) \quad \Gamma, f \colon \langle T \rangle \vdash Q \colon (\mathsf{end}, T')}{\Gamma \vdash \mathsf{stream}\, P\, \mathsf{as}\, f\, \mathsf{in}\, Q \colon (U, T')}$$

input within a session

service definition

Sample rules

input
within a
session

service
definition

parallel
composition

$$\frac{\Gamma, x\colon T' \vdash P\colon (U, T)}{\Gamma \vdash (x)P\colon (?T'.U, T)}$$

$$\frac{\Gamma \vdash P\colon (U, T) \quad \Gamma \vdash a\colon [U]}{\Gamma \vdash a \Rightarrow P\colon (\mathsf{end}, T)}$$

$$\frac{\Gamma \vdash P\colon (U, T) \quad \Gamma \vdash Q\colon (\mathsf{end}, T)}{\Gamma \vdash P|Q\colon (U, T)}$$

$$\frac{\Gamma \vdash P\colon (U, T) \quad \Gamma, f\colon \langle T \rangle \vdash Q\colon (\mathsf{end}, T')}{\Gamma \vdash \mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q\colon (U, T')}$$

Sample rules

$$\frac{\Gamma, x\colon T' \vdash P\colon (U,T)}{\Gamma \vdash (x)P\colon (?T'.U,T)}$$

input within a session

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash a\colon [U]}{\Gamma \vdash a \Rightarrow P\colon (\mathsf{end},T)}$$

service definition

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash Q\colon (\mathsf{end},T)}{\Gamma \vdash P|Q\colon (U,T)}$$

parallel composition

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma, f\colon \langle T\rangle \vdash Q\colon (\mathsf{end},T')}{\Gamma \vdash \mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q\colon (U,T')}$$

(empty) stream

Type safety

- Subject reduction

types for the free identifiers

If $\Gamma \vdash P : (U, T)$ and $P \rightarrow P'$, then $\Gamma \vdash P' : (U, T)$

- Type safety

  "Well typed programs do not go wrong"

thread-sync
+
intra-thread comm

Further analyses

- Program equivalence (mentioned before)
  - congruence; axiomatic laws
- Deadlock avoidance:
  - communication errors within a session (addressed before)
  - no service for a particular consumer (several proposals in process calculi)
  - read from an empty stream (see paper)

Summary

- Presented language

  “Stream-based Service Centered Calculus”

  describing services, conversations, and orchestration

- Amenable to different sort of analyses
- Encoded all van der Aalst workflow patterns -> expressiveness “test”

Future

- Develop bisimulation techniques
- Extend the language with some form of failure/exception and corresponding **compensation** mechanism

http://www.sensoria-ist.eu/