Faster Coroutine Pipelines: A Reconstruction

Ruben Pieters Tom Schrijvers

KU LEUVEN

Stream Processing

Data → Stream Processing

- Large amounts
- Handle efficiently

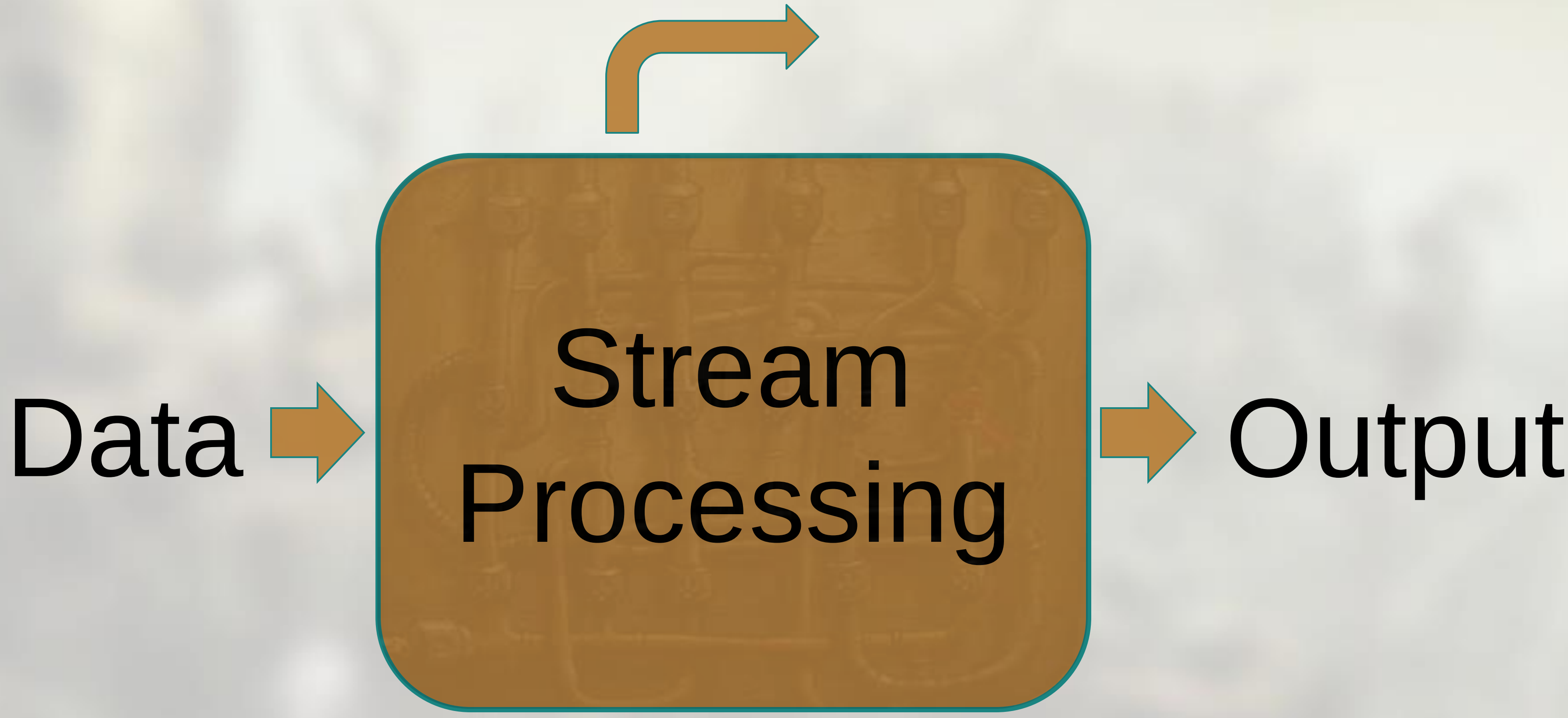
Data

Stream Processing

Output

- End result

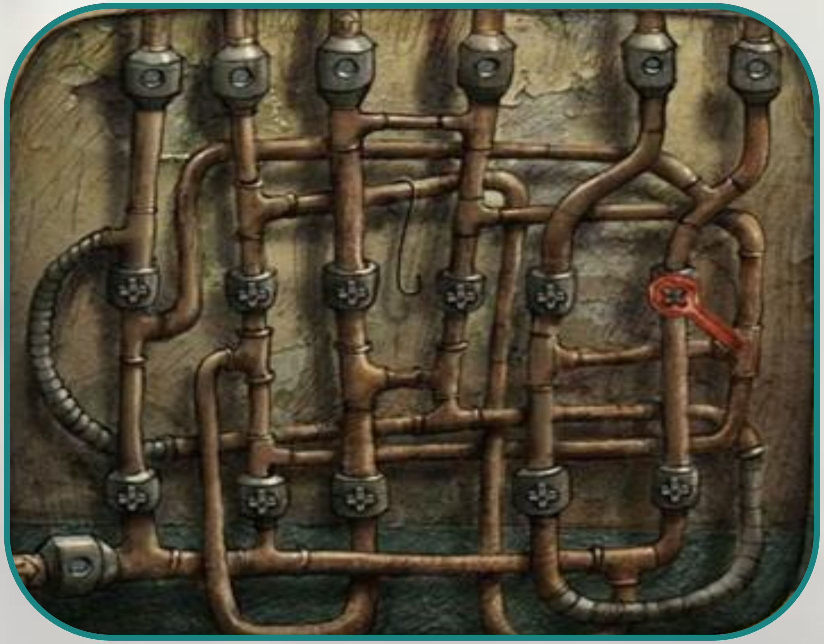
Side-Effects



- Example: read from file, write to database...

Side-Effects

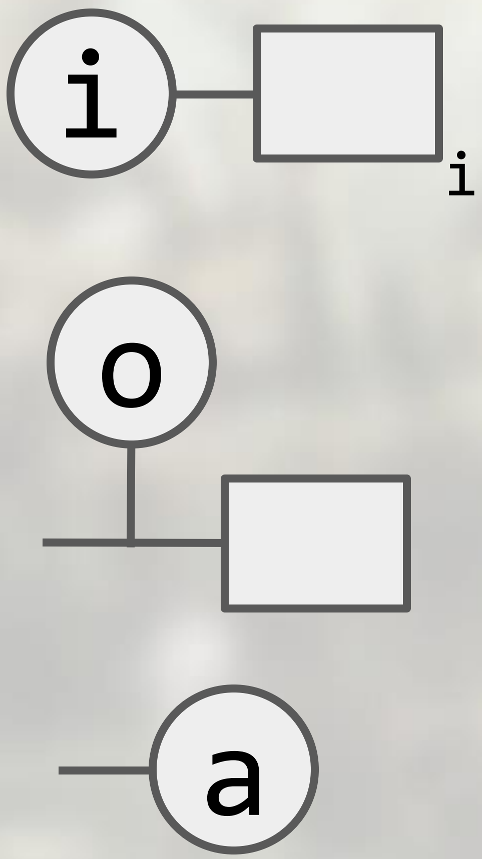
Data



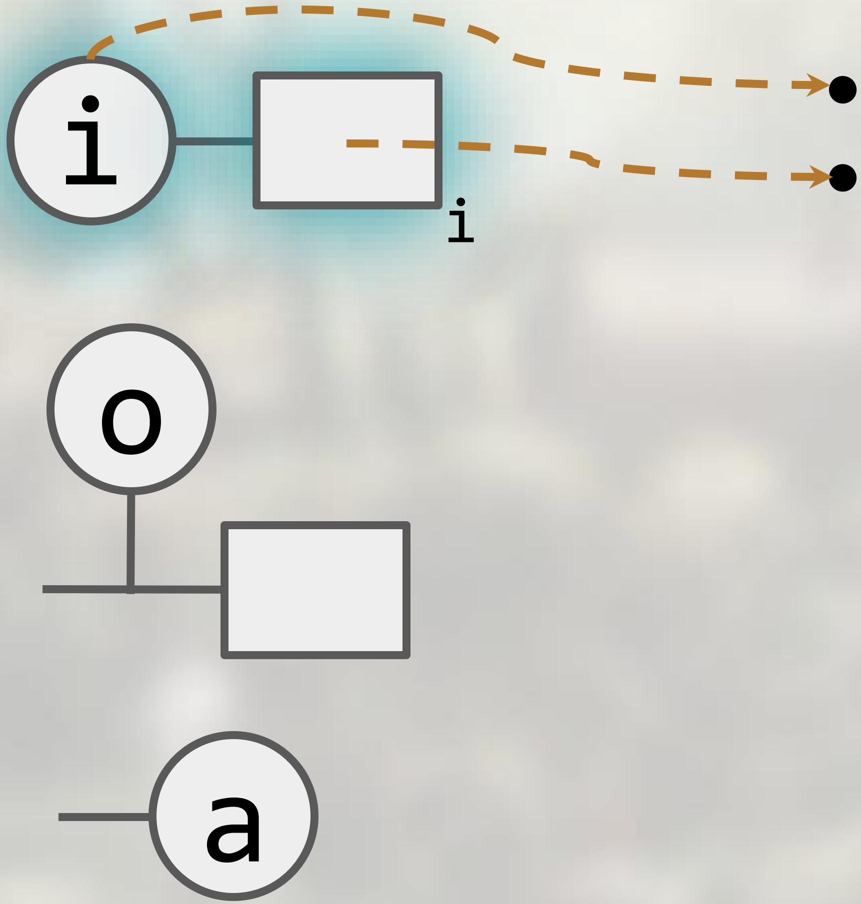
Output

- **Pipes** and **Three-Continuation Approach**
  `Faster Coroutine Pipelines` by Spivey
  `Continuations and Transducer Composition` by Shivers and Might

Basic Building Blocks

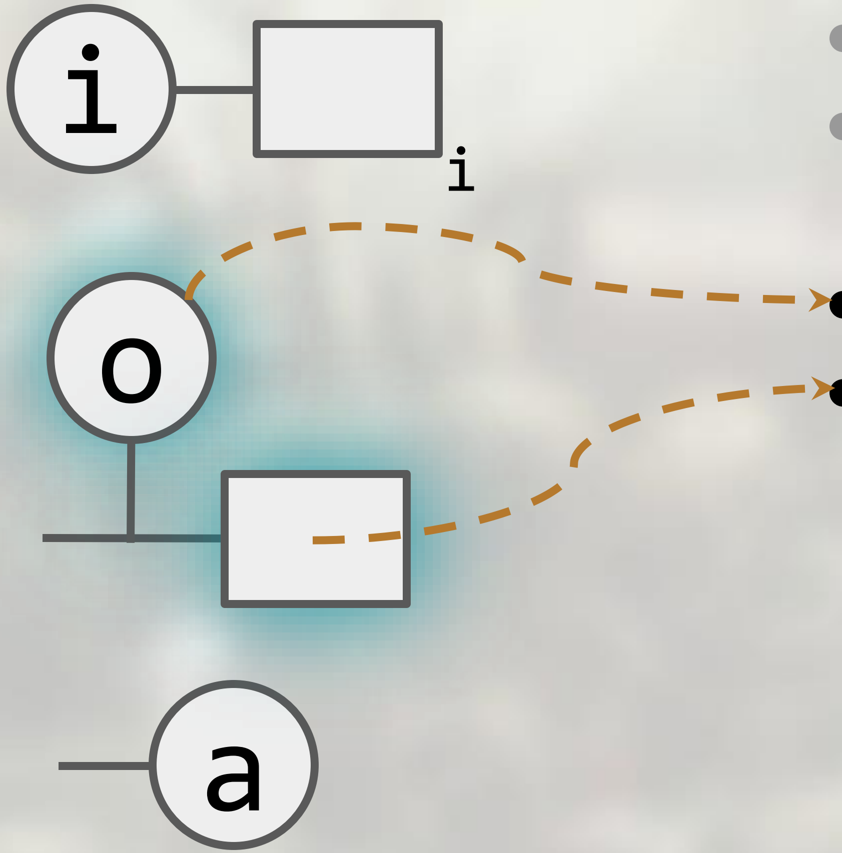


Basic Building Blocks: Input



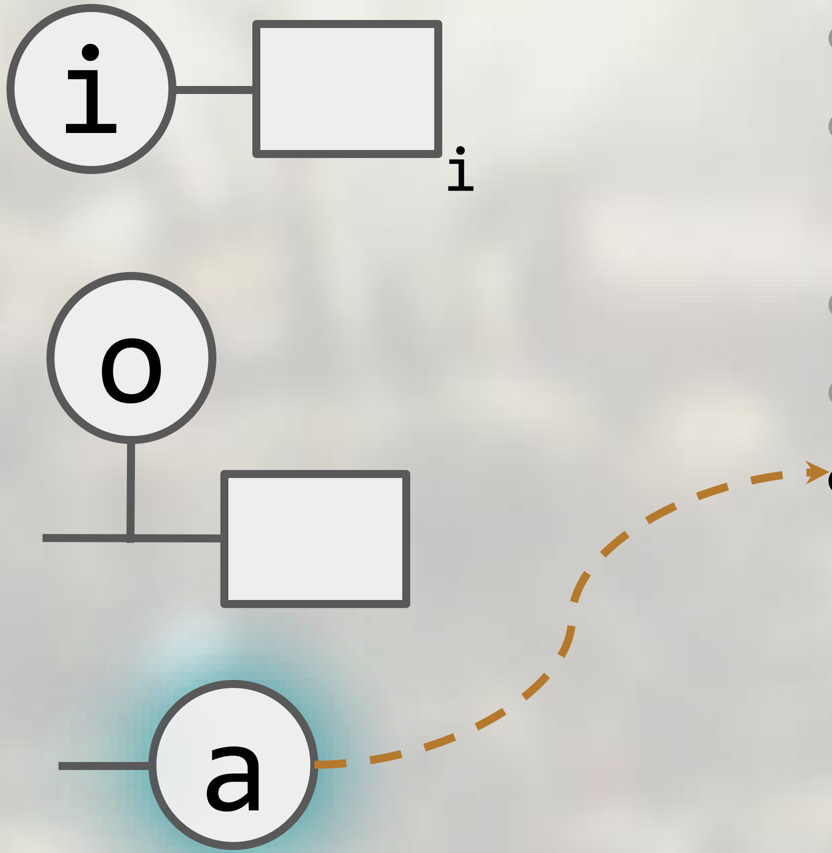
- Take a value **`i` as input**
- The rest of the pipeline, which **can depend on `i`**

Basic Building Blocks: Output



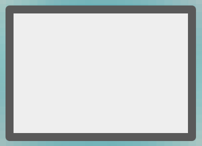
- Take a value `i` **as input**
- The rest of the pipeline, which **can depend on `i`**
- **Output an `o`** value
- The rest of the pipeline

Basic Building Blocks: Return

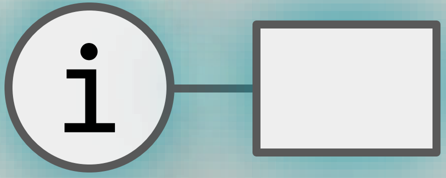


- Take a value `i` **as input**
- The rest of the pipeline, which **can depend on `i`**
- **Output an `o`** value
- The rest of the pipeline
- **Return an `a` value**

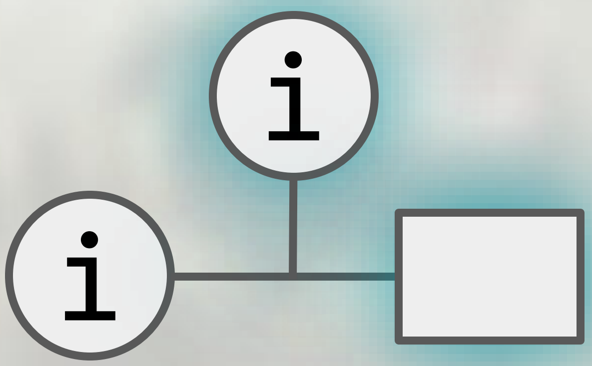
Example: Building a Pipeline



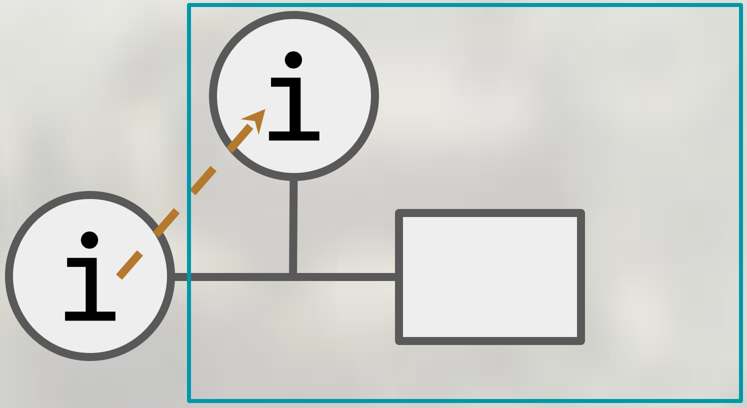
Example: Building a Pipeline



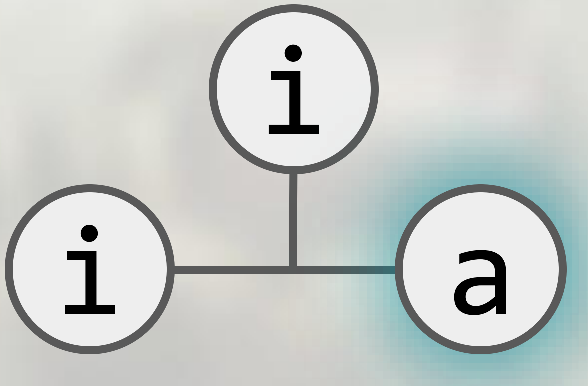
Example: Building a Pipeline



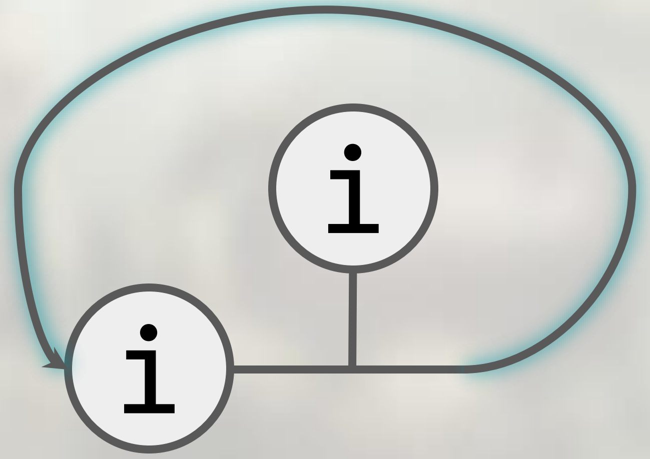
Example: Building a Pipeline



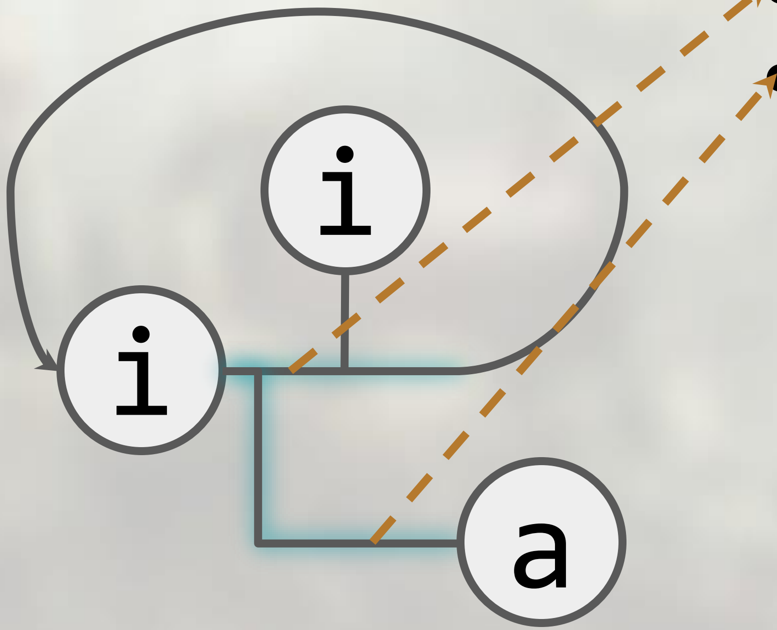
Example: Building a Pipeline



Example: Building a Pipeline

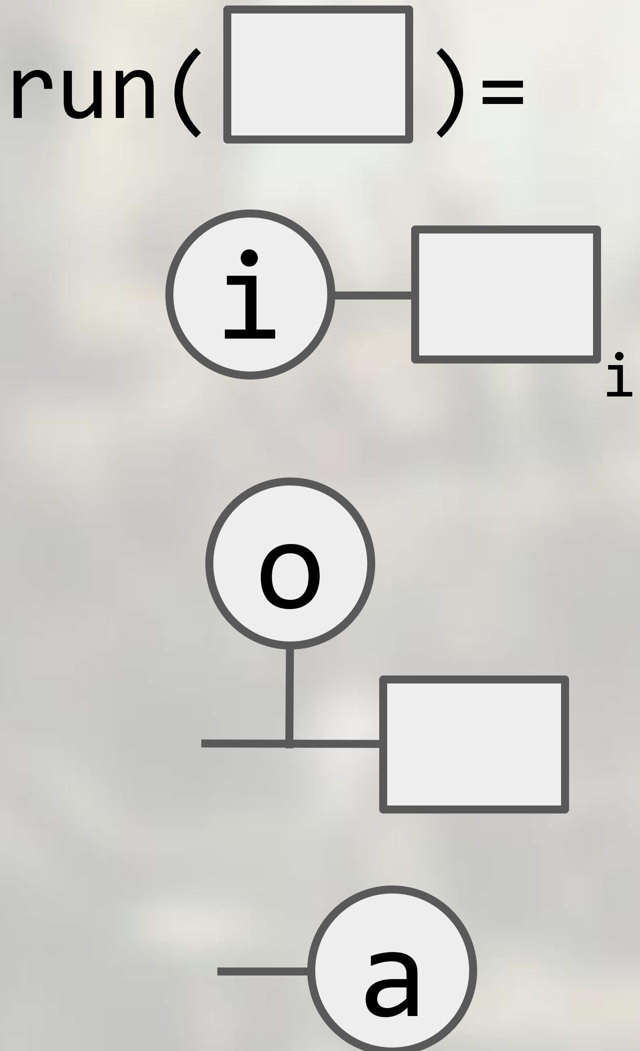


Example: Building a Pipeline

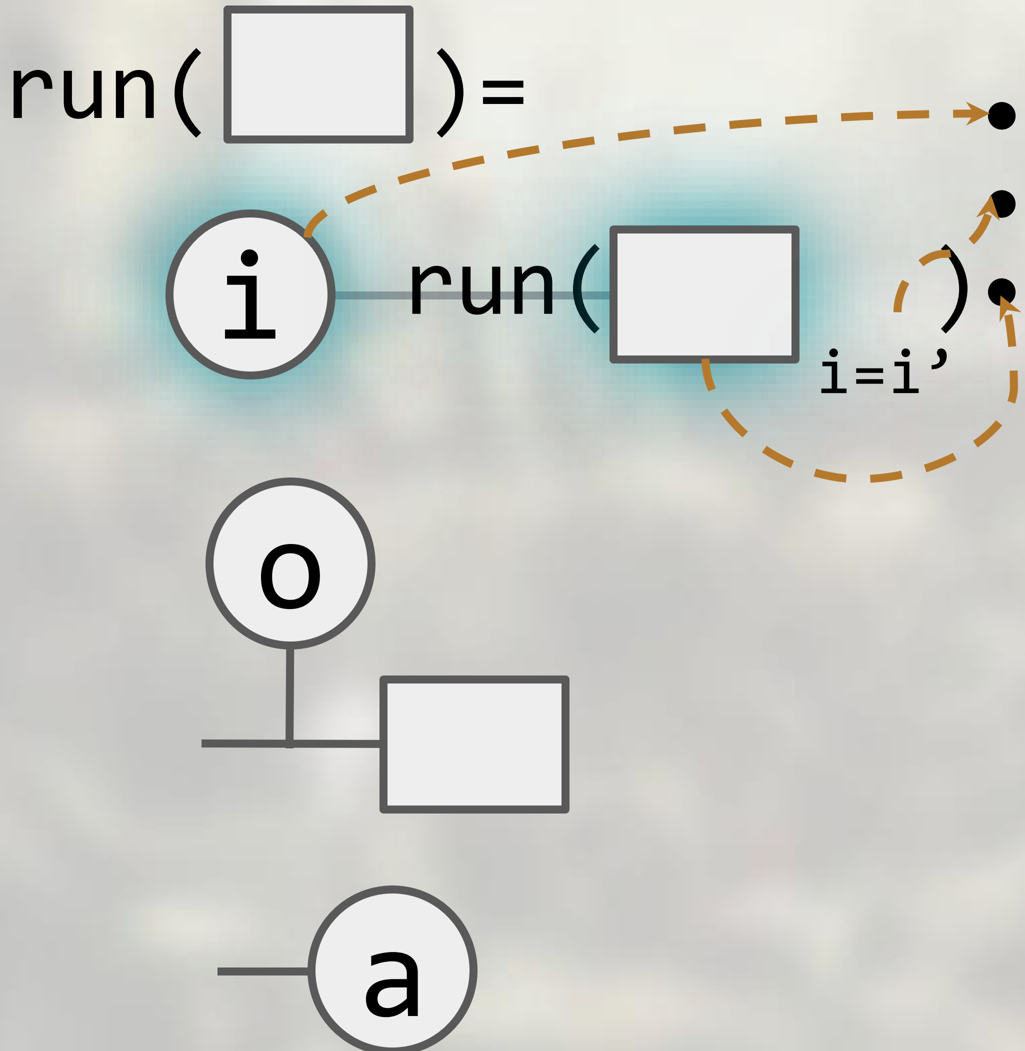


- If i > 0
- If i ≤ 0

Running a Pipeline

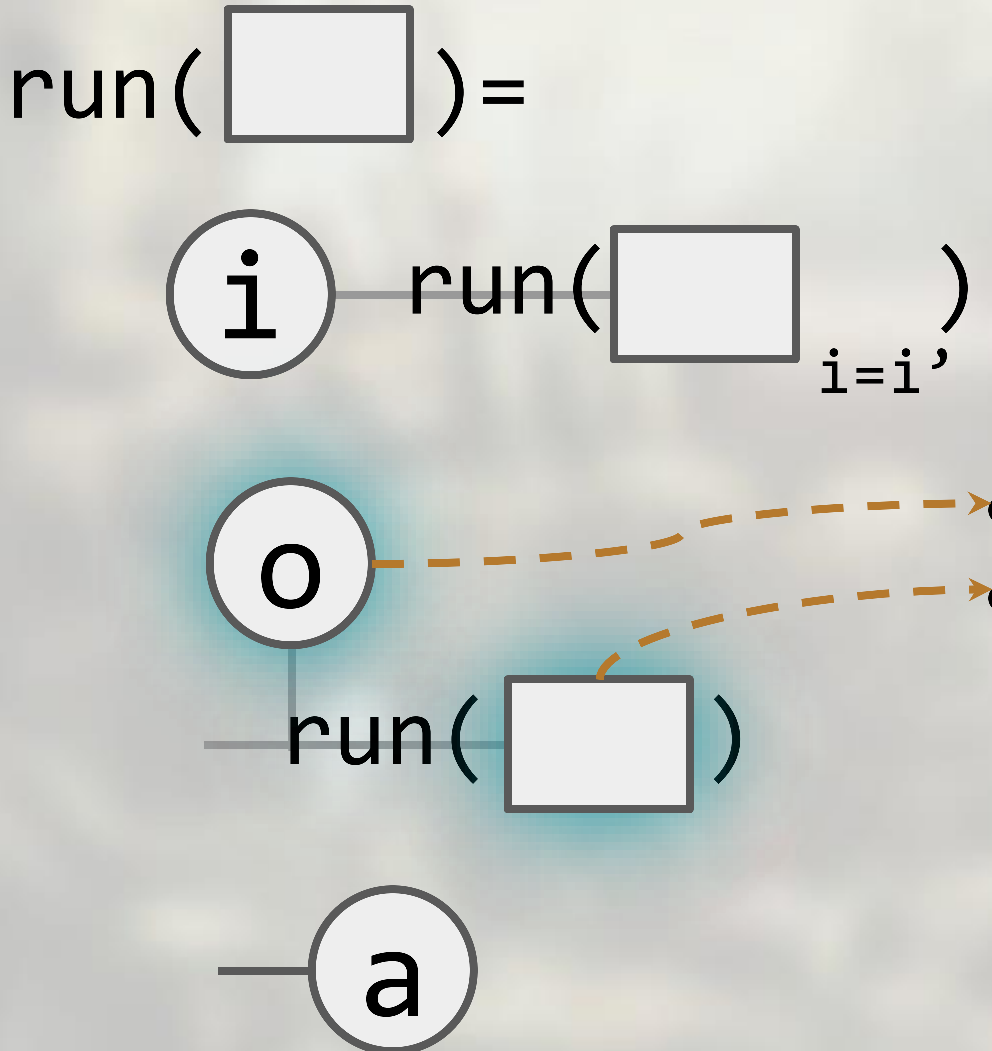


Running a Pipeline



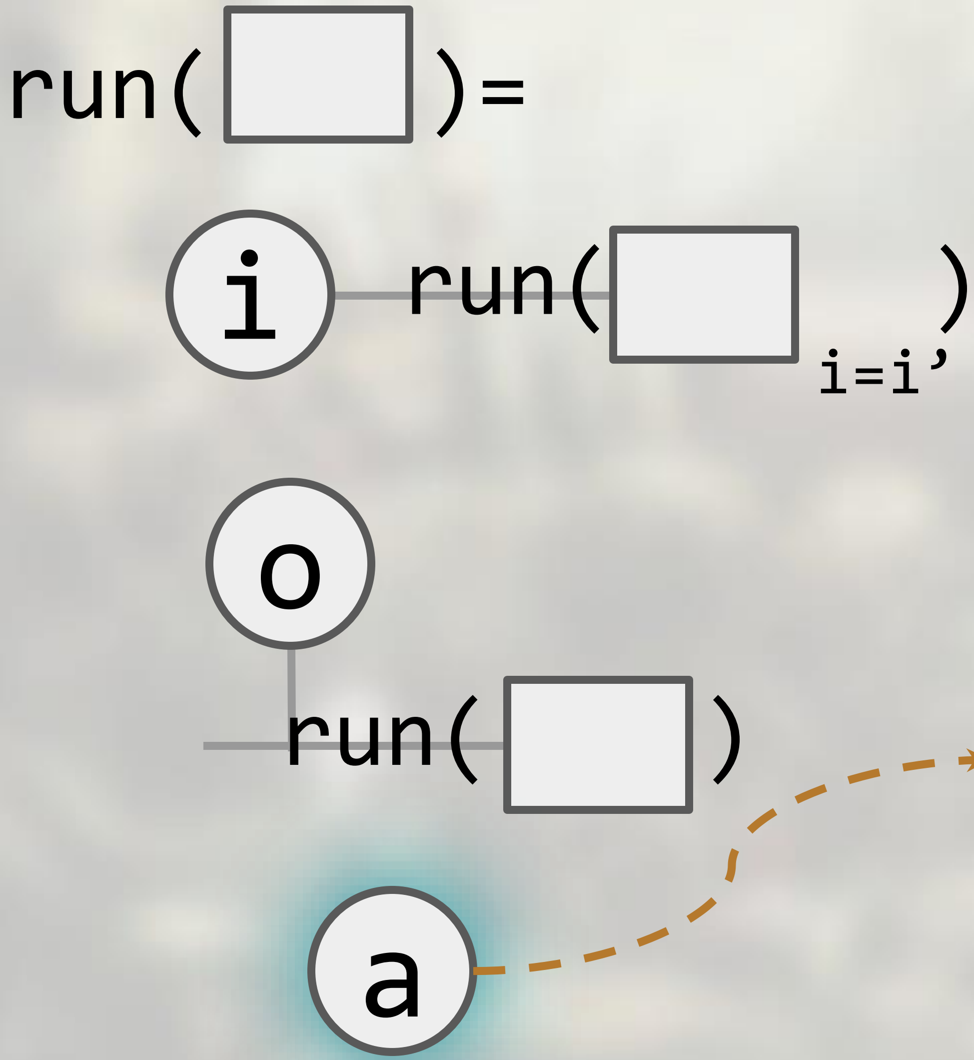
- Ask user for value `i’`
- Substitute `i` with `i’`
- Run the rest

Running a Pipeline



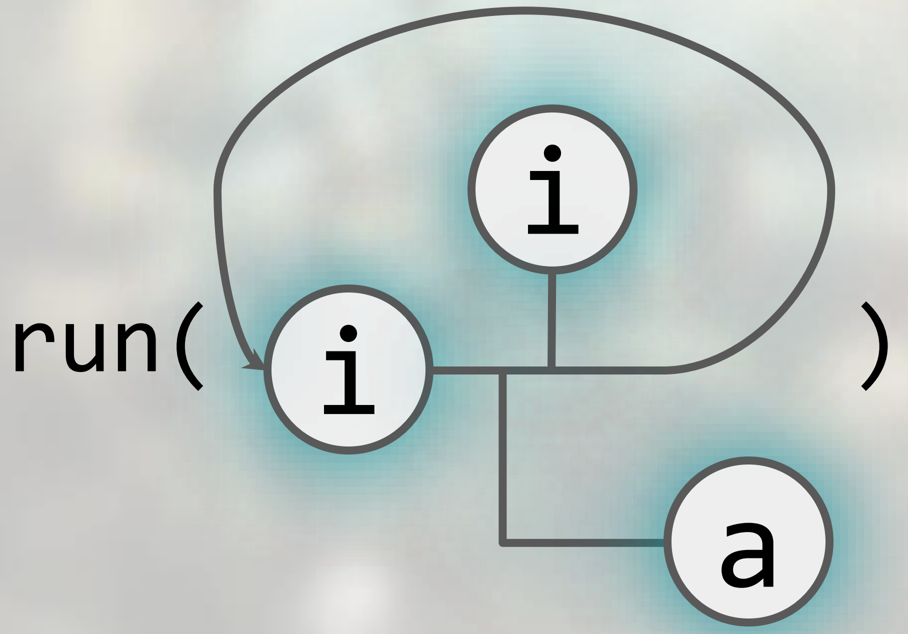
- Print value o
- Run the rest

Running a Pipeline



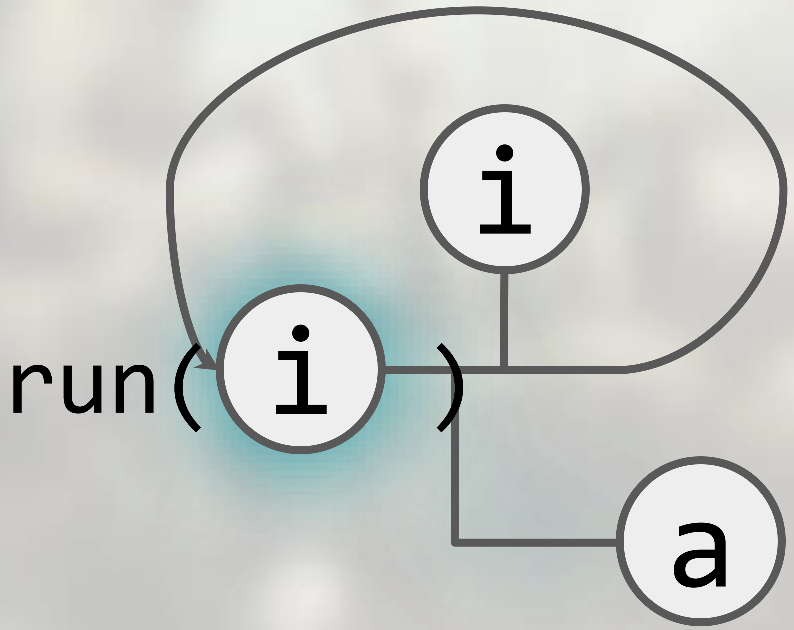
- Return value a

Example: Running a Pipeline



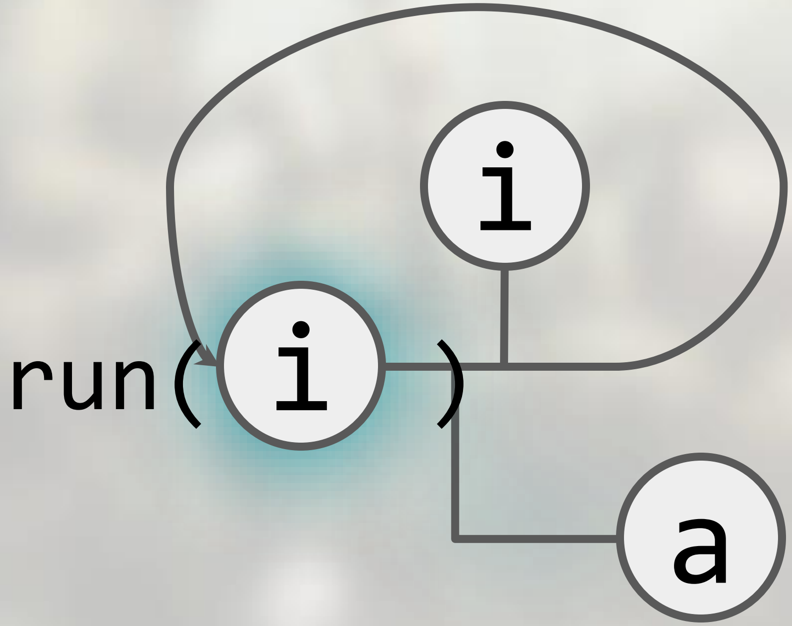

```
> run example
```

Example: Running a Pipeline



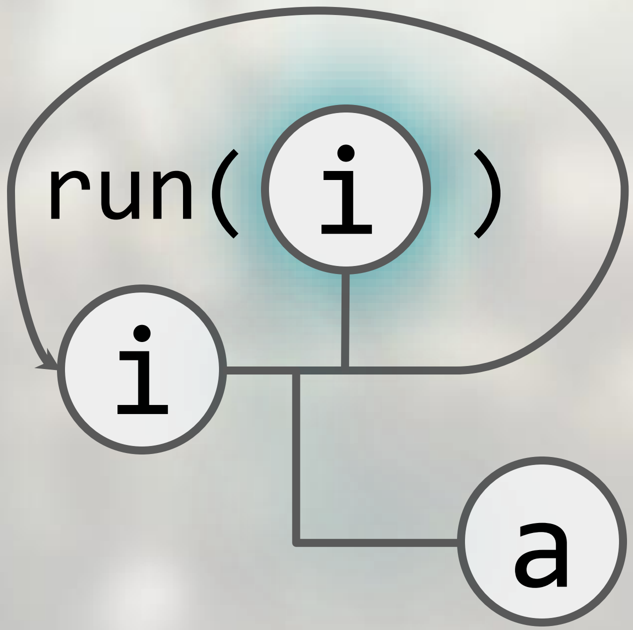

```
> run example
input:
```

Example: Running a Pipeline



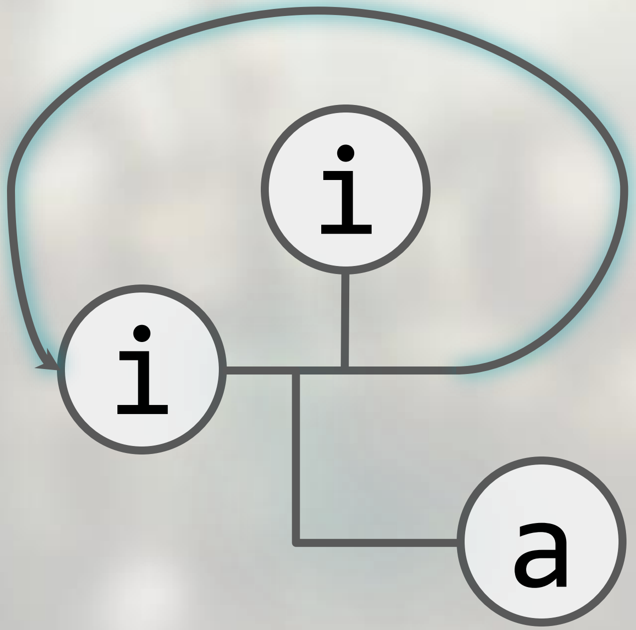

```
> run example
input:
42
```

Example: Running a Pipeline



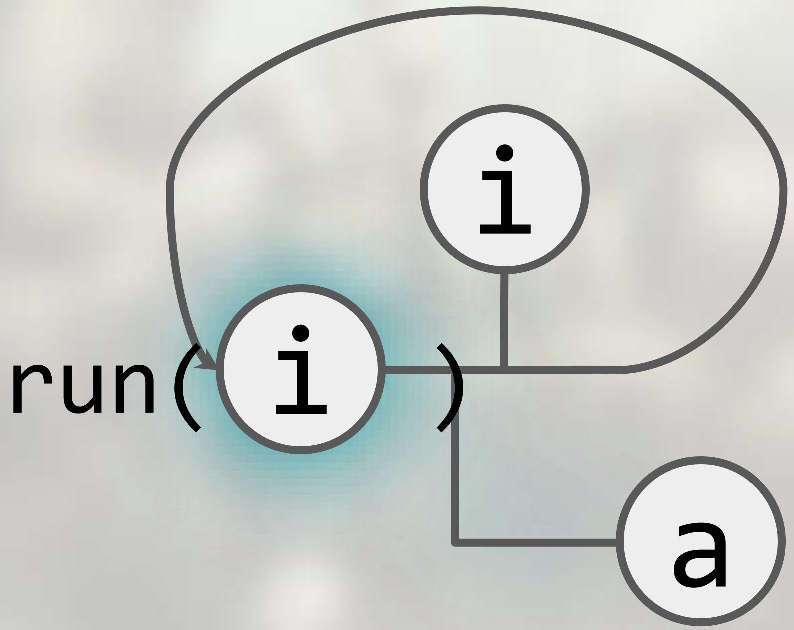

```
> run example
input:
42
output: 42
```

Example: Running a Pipeline



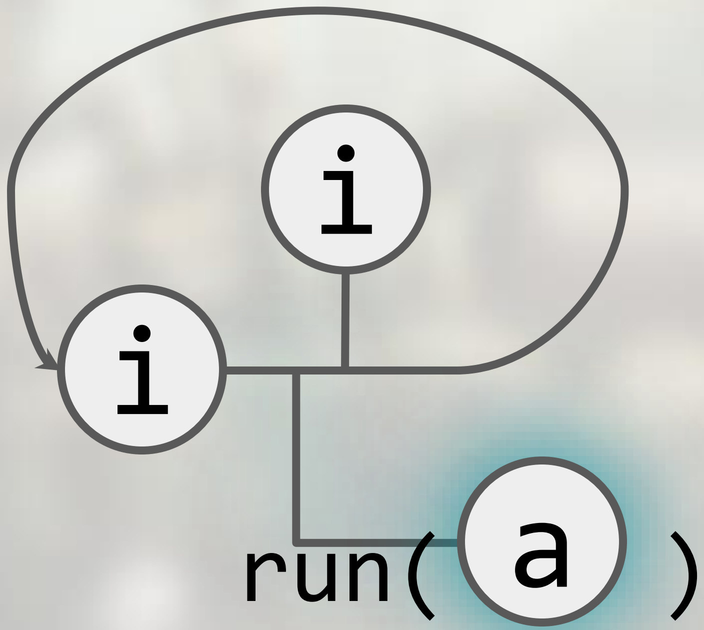

```
> run example
input:
42
output: 42
```

Example: Running a Pipeline




```
> run example
input:
42
output: 42
input:
-1
```

Example: Running a Pipeline




```
> run example
input:
42
output: 42
input:
-1
return: a
Pipeline finished
>
```

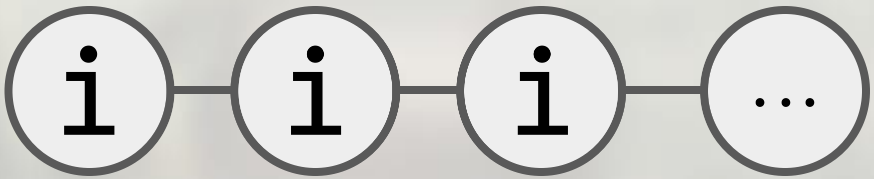
Pipelines as Building Blocks

Pipelines as Building Blocks

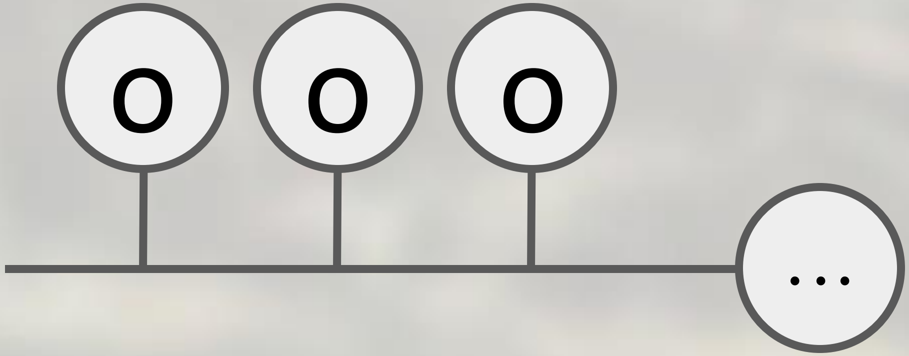
```
merge(□ , □) = □
```

One-Sided Pipes

Consumer:



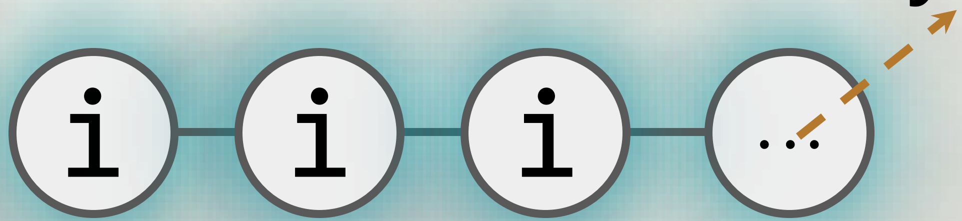
Producer:



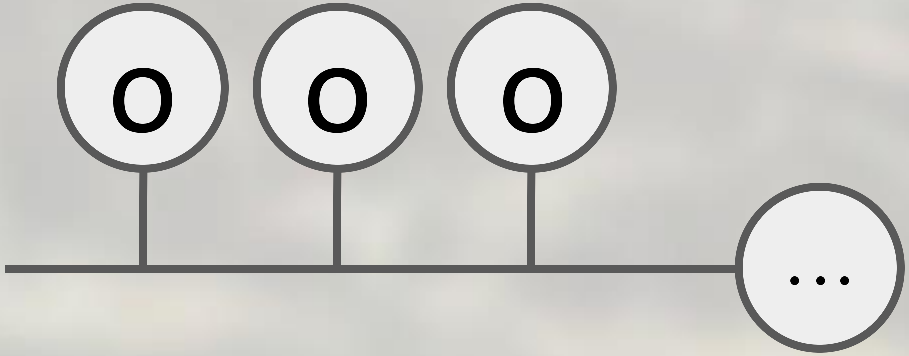
One-Sided Pipes

Consumer:

Only Input

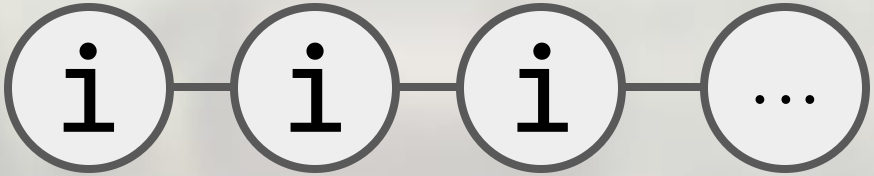


Producer:



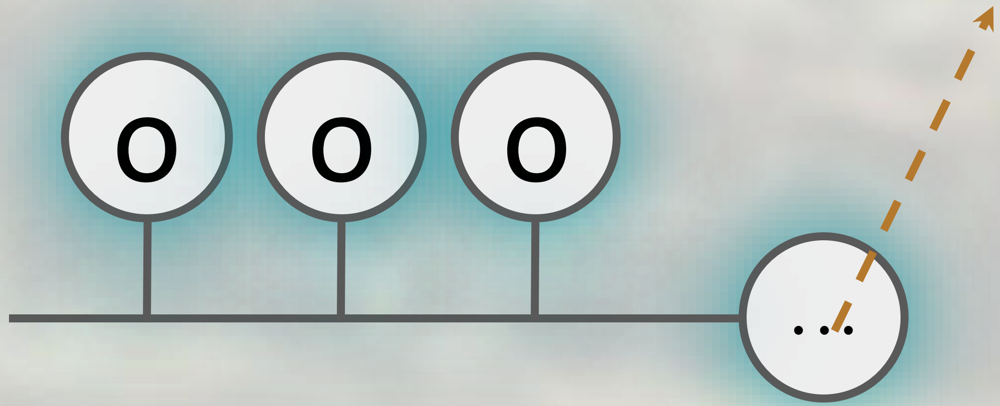
One-Sided Pipes

Consumer:

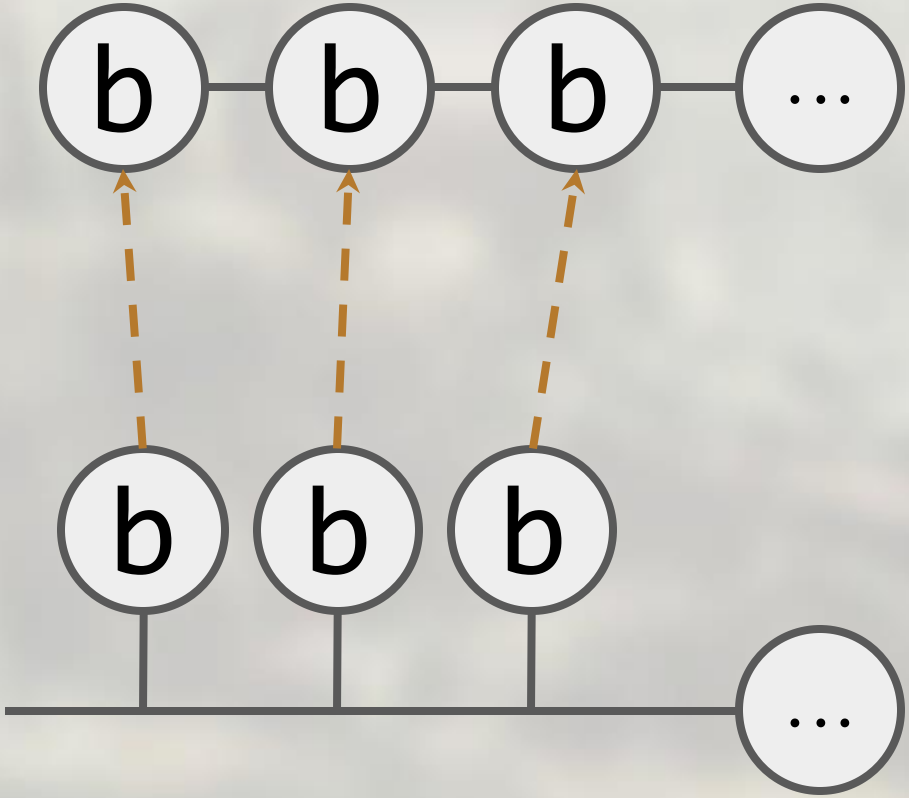


Producer:

Only Output



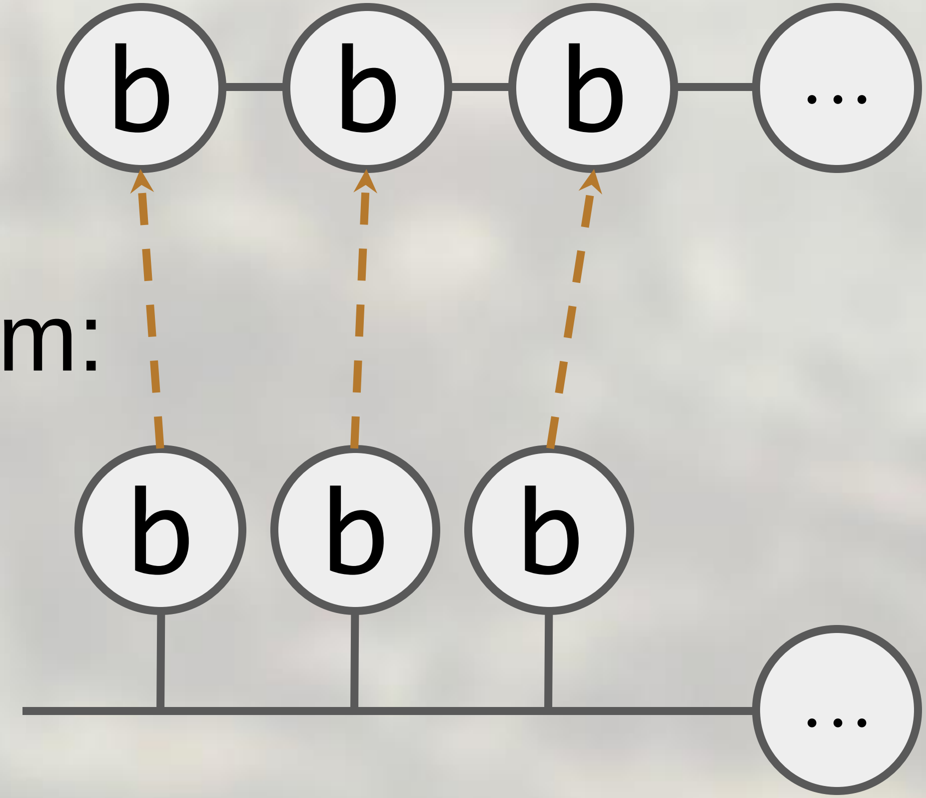
One-Sided Pipes: Merge Example



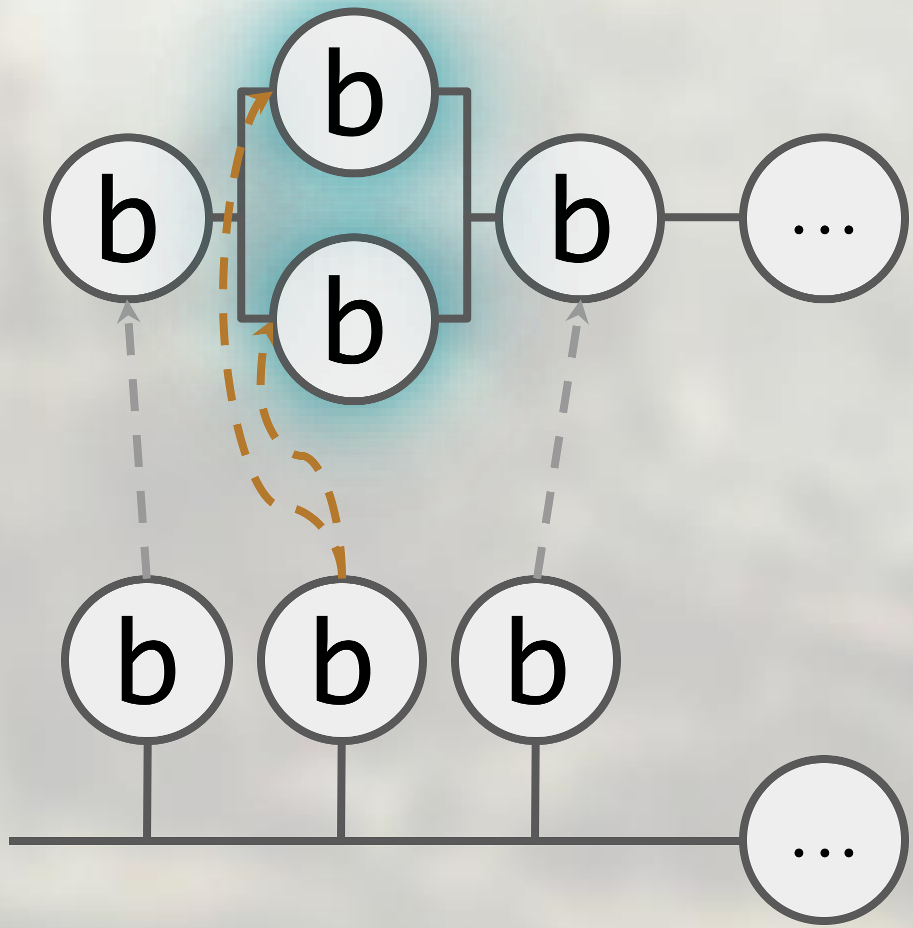
One-Sided Pipes: Merge Example

Downstream:

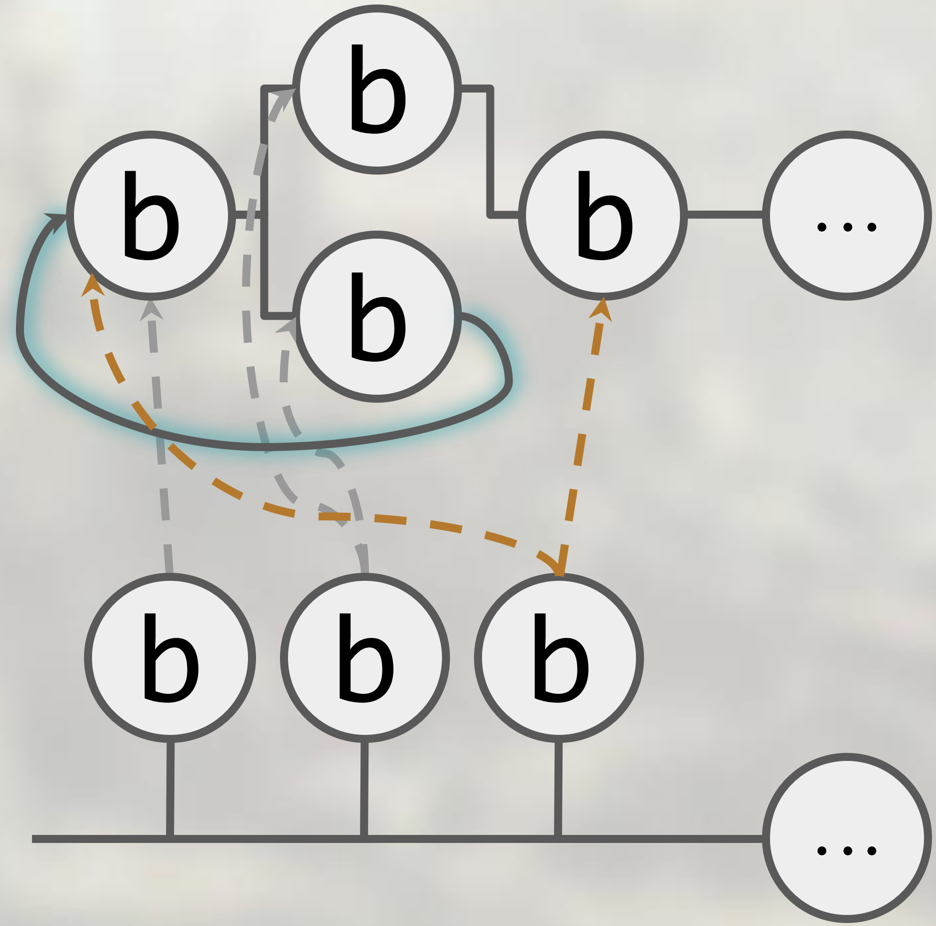
Upstream:



One-Sided Pipes: Merge Example

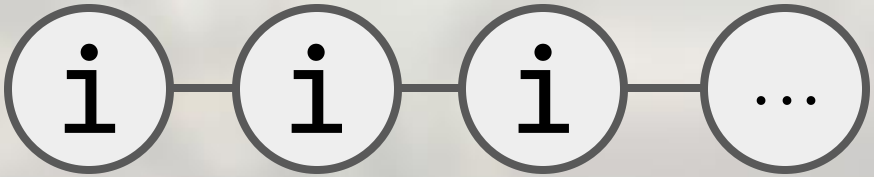


One-Sided Pipes: Merge Example

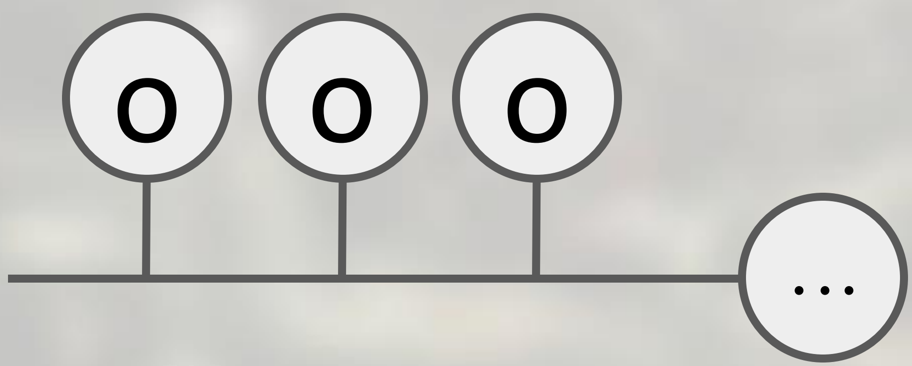


Representation

Consumer:



Producer:



Representation

Consumer:

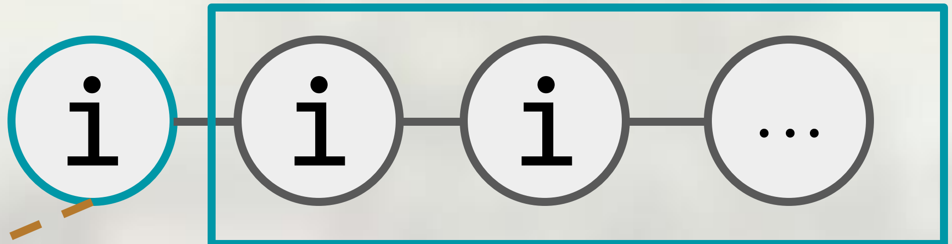
```
data Consumer i = C (i -> Consumer i)
C (\i₀ -> C (\i₁ -> C (...)))
```

Producer:

```
data Producer o = P o (Producer o)
P o₀ (P o₁ (P o₂ (...)))
```

Representation

Consumer:




```
data Consumer i = C (i -> Consumer i)
C (\i0 -> C (\i1 -> C (...)))
```

Producer:

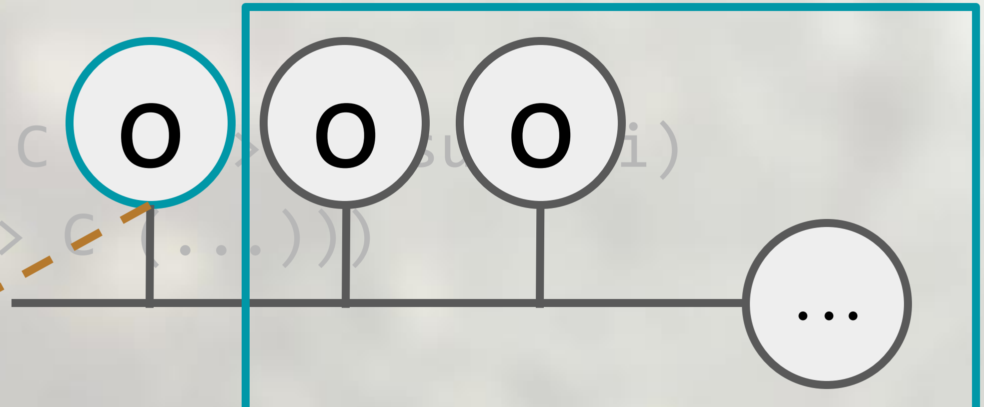
```
data Producer o = P o (Producer o)
P o0 (P o1 (P o2 (...)))
```

Representation

Consumer:

```
data Consumer i = C (i -> Consumer i)
```

$\texttt{C (\textbackslash i}_0\texttt{ -> C (\textbackslash i}_1\texttt{ -> C (...)))}$



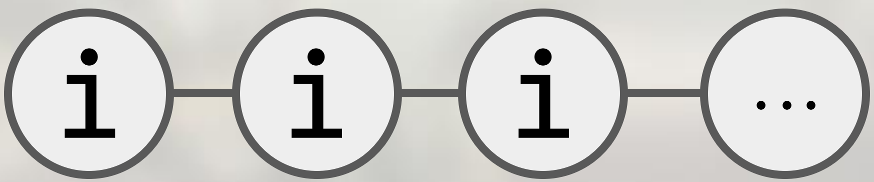
Producer:

```
data Producer o = P o (Producer o)
```

$\texttt{P}\ \boxed{\texttt{o}_0}\ \boxed{\texttt{(P o}_1\texttt{ (P o}_2\texttt{ (...)))}}$

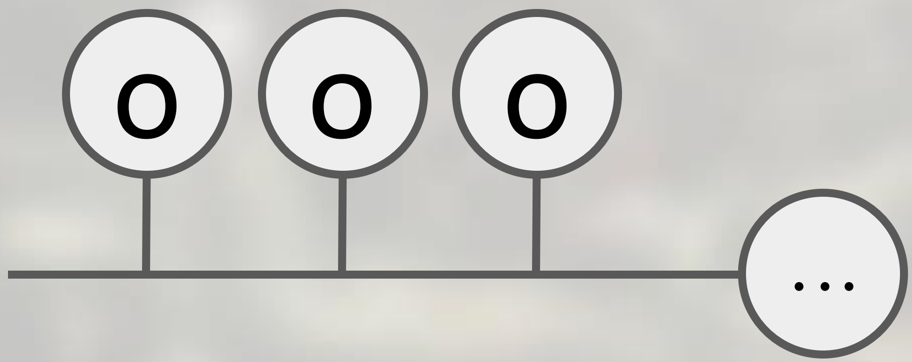
Representation

Consumer:



`: Function`

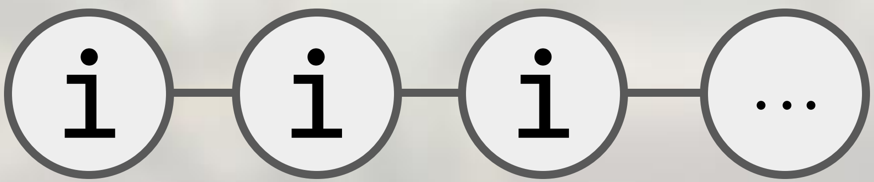
Producer:



`: Function`

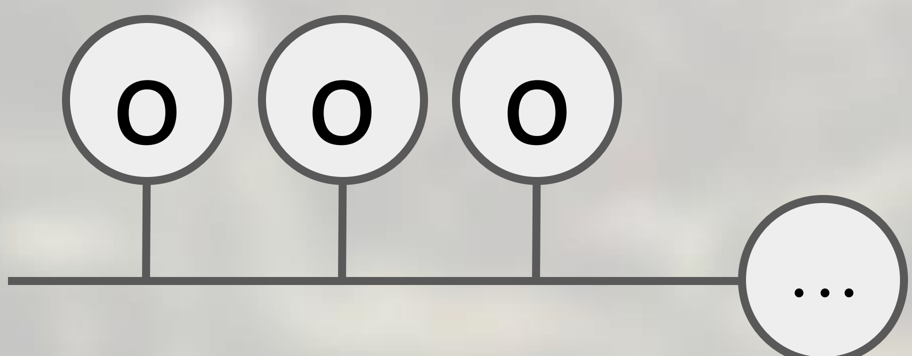
Representation

Consumer:



Church

Producer:



Representation

Consumer:

```
\k -> k (\i₀ -> k (\i₁ -> k (...)))
```

Producer:

```
\k -> k o₀ (k o₁ (k o₂ (...)))
```

Representation

Consumer:

run

```
\k -> k (\i_0 -> k (\i_1 -> k (...)))
```

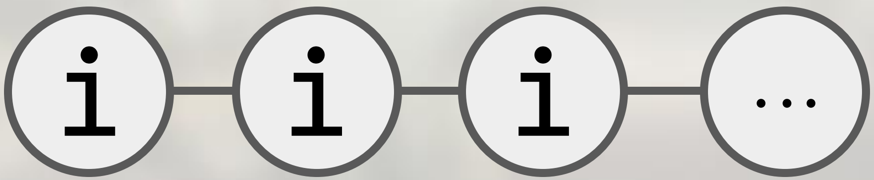
Producer:

run

```
\k -> k o_0 (k o_1 (k o_2 (...)))
```

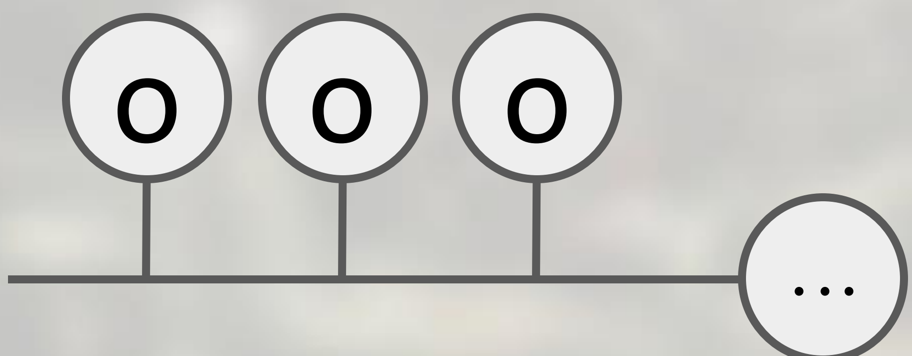
Representation

Consumer:



Scott

Producer:



Representation

Consumer:

$$\backslash k_0 \ \texttt{->}\ k_0\ \boxed{(\backslash i_0}\ \boxed{k_1\ \texttt{->}\ k_1\ (\backslash i_1\ k_2\ \texttt{->}\ k_2\ (...)))}$$

Producer:

$$\backslash k_0 \ \texttt{->}\ k_0\ \boxed{o_0}\ \boxed{(\backslash k_1\ \texttt{->}\ k_1\ o_1\ (\backslash k_2\ \texttt{->}\ k_2\ o_2\ (...)))}$$

Representation

Consumer:

run run run

$$\texttt{\textbackslash k}_0\ \texttt{->}\ \texttt{k}_0\ \texttt{(\textbackslash i}_0\ \texttt{k}_1\ \texttt{->}\ \texttt{k}_1\ \texttt{(\textbackslash i}_1\ \texttt{k}_2\ \texttt{->}\ \texttt{k}_2\ \texttt{(...)))}$$

Producer:

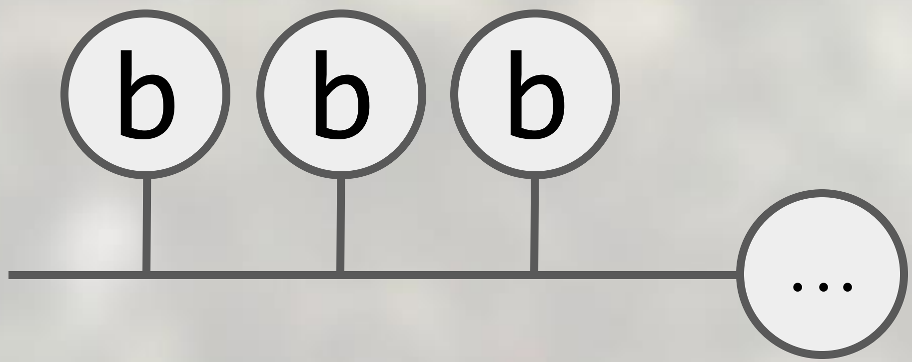
run run run

$$\texttt{\textbackslash k}_0\ \texttt{->}\ \texttt{k}_0\ \texttt{o}_0\ \texttt{(\textbackslash k}_1\ \texttt{->}\ \texttt{k}_1\ \texttt{o}_1\ \texttt{(\textbackslash k}_2\ \texttt{->}\ \texttt{k}_2\ \texttt{o}_2\ \texttt{(...)))}$$

Representation

```
merge = apply(
```


```
,
```

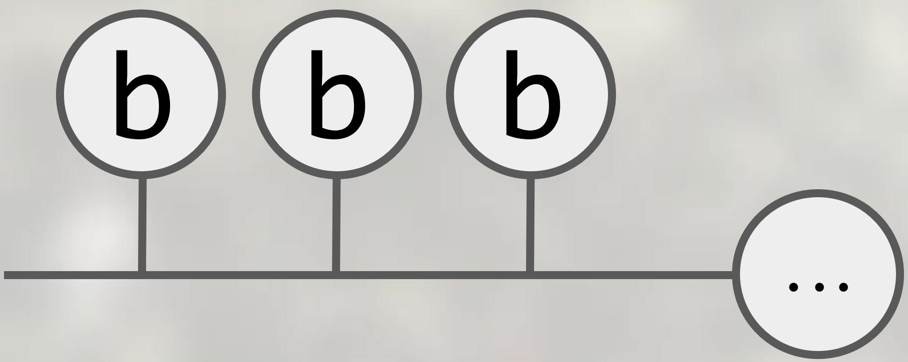



```
)
where apply(f,x) = f x
```

Representation

```
merge = apply(
```


```
,
```



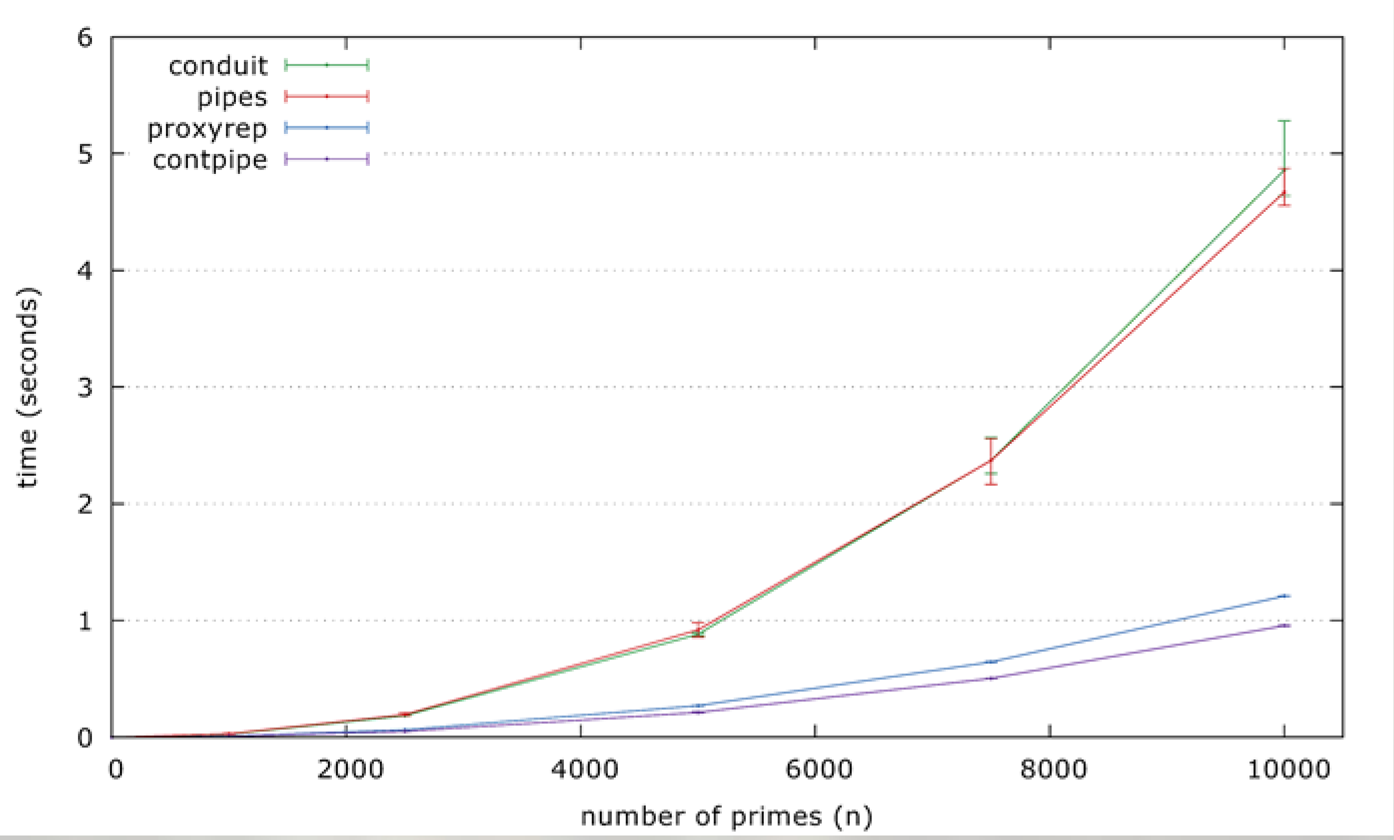

```
)
where apply(f,x) = f x
```

*Haskell: modulo **newtype** wrappers

The Paper

Presents a stepwise approach to arrive at this representation.

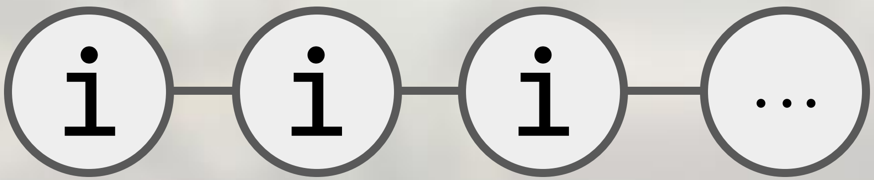
The Paper



conduit
pipes
proxyrep
contpipe
time (seconds)
number of primes (n)
0
1
2
3
4
5
6
0
2000
4000
6000
8000
10000

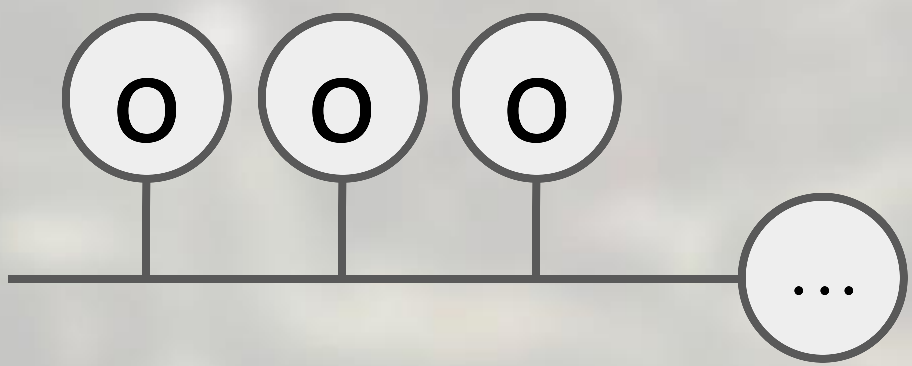
Infinite Pipes

Consumer:



$: K_i \rightarrow A$

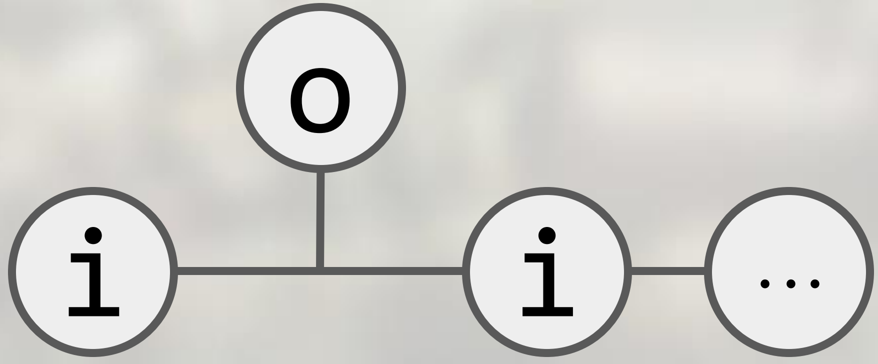
Producer:



$: K_o \rightarrow A$

Infinite Pipes

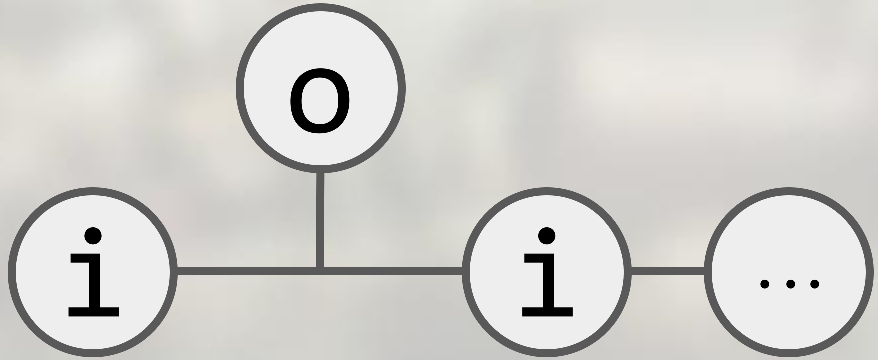
Pipe_∞:



$: K_i \to K_o \to A$

Infinite Pipes

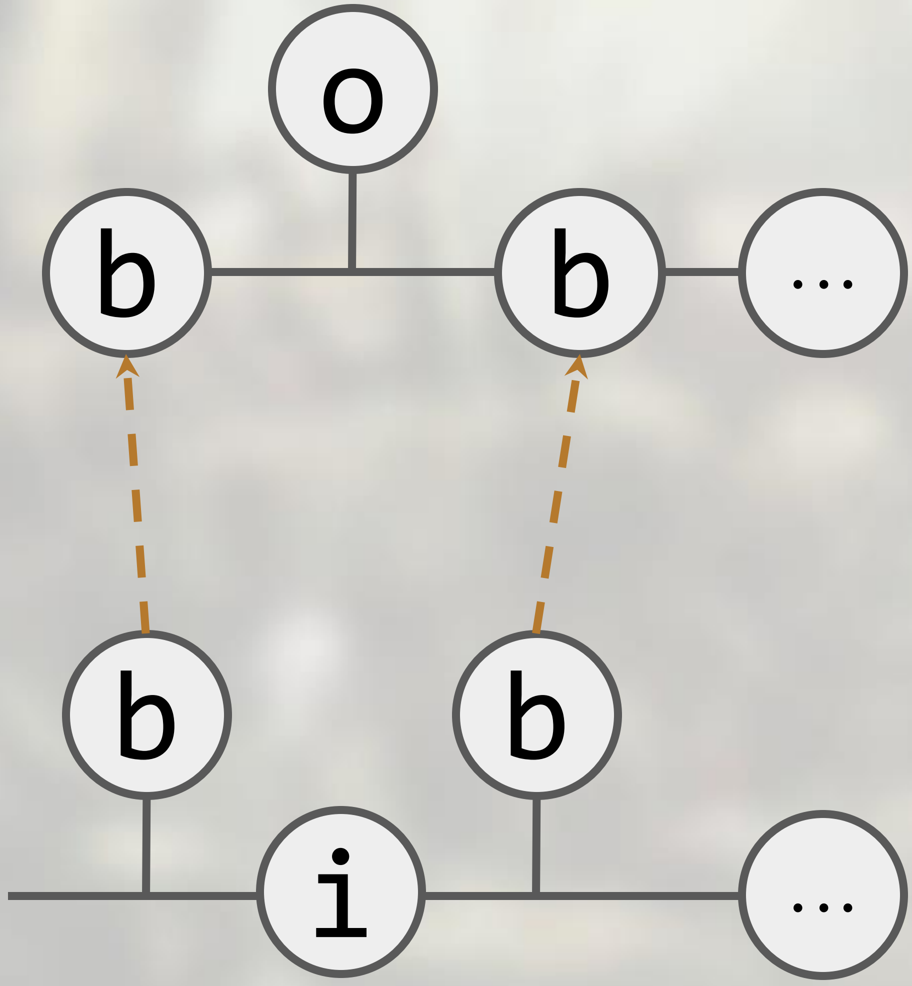
Pipe_∞:



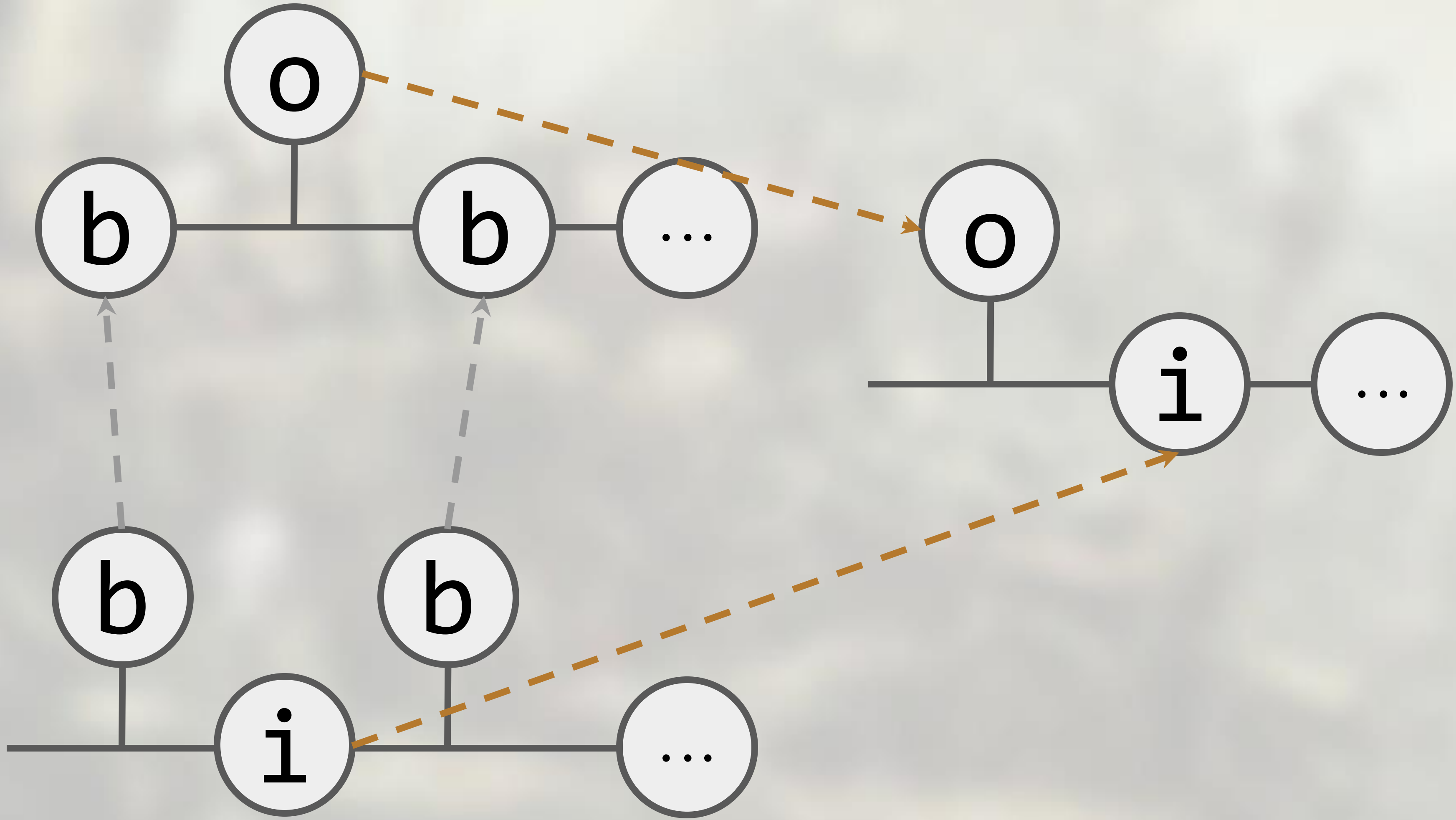
$: K_i \to K_o \to A$

(Not Scott Encoding)

Merge Example

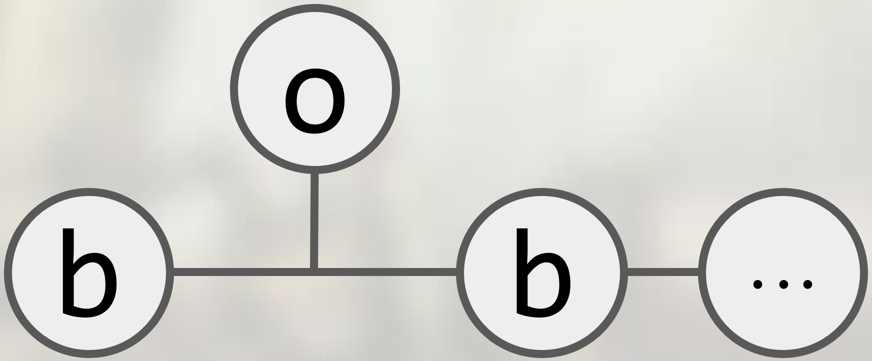


Merge Example

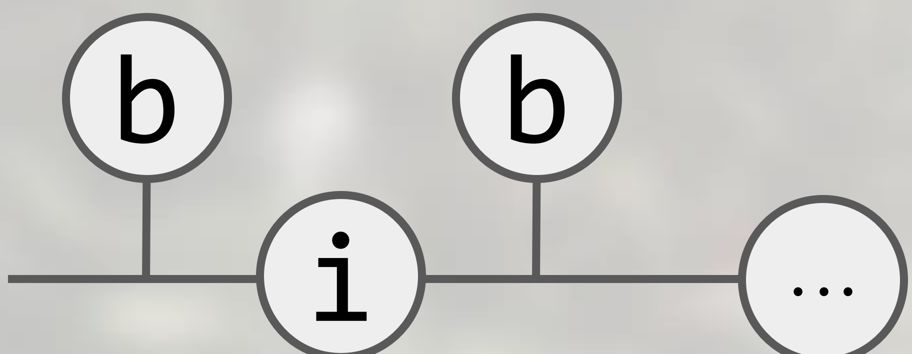


Infinite Pipes: Merge

K_i -> K_o -> A

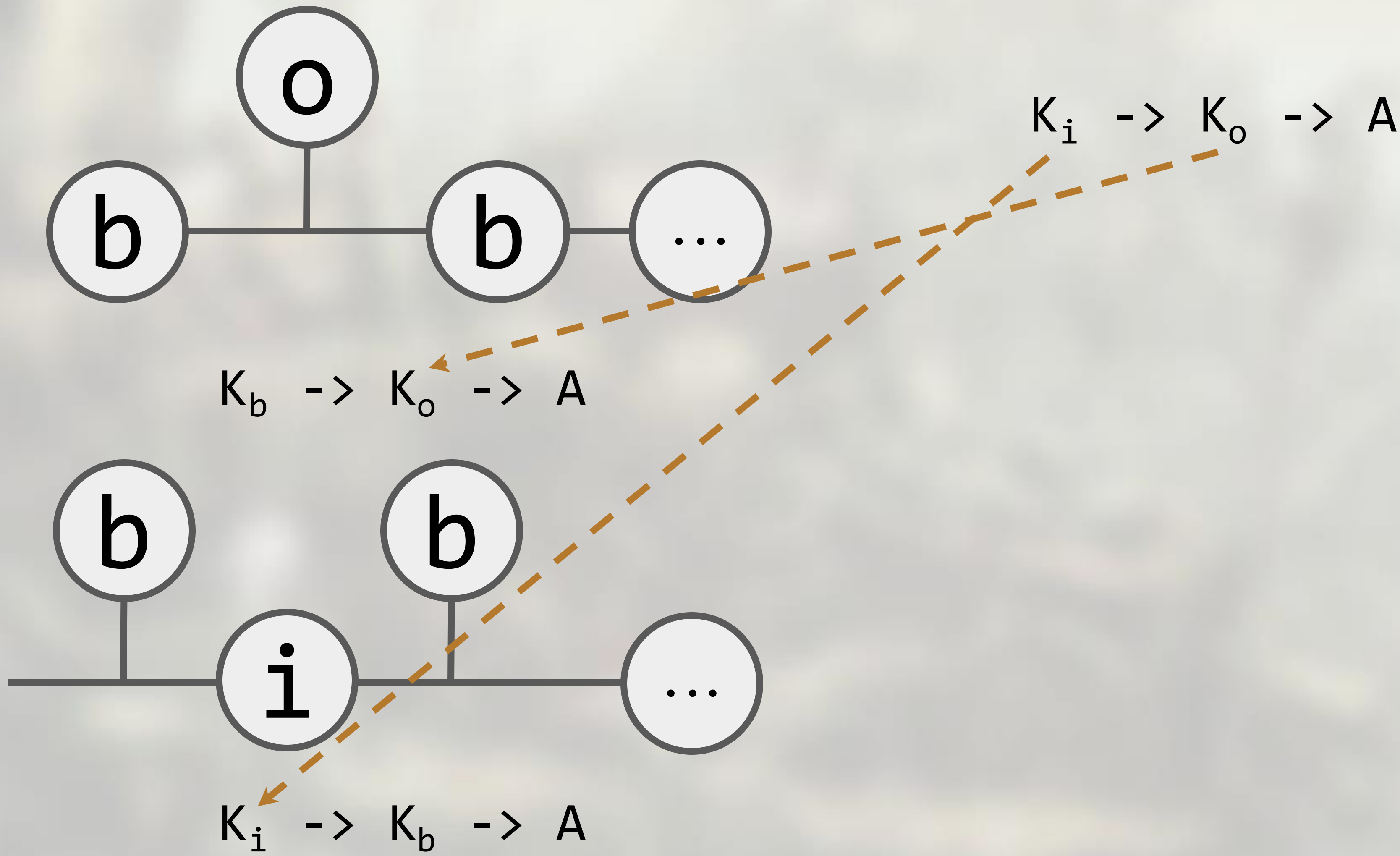


K_b -> K_o -> A

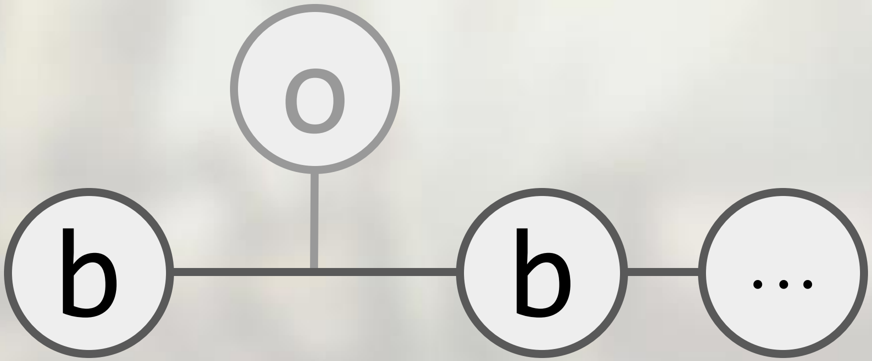


K_i -> K_b -> A

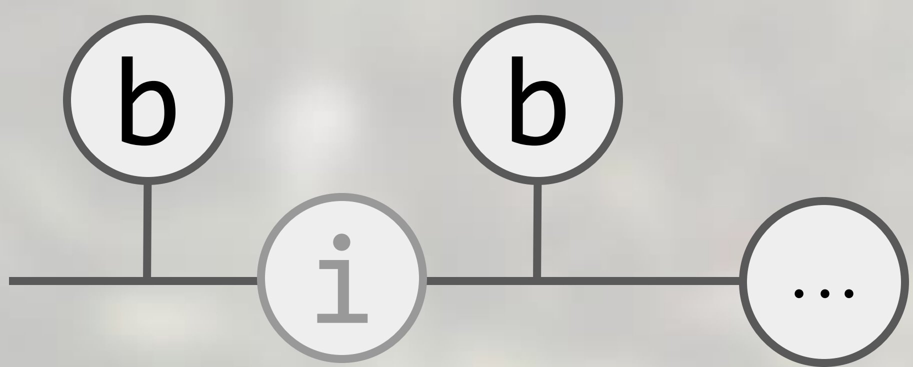
Infinite Pipes: Merge



Infinite Pipes: Merge



K_b -> A

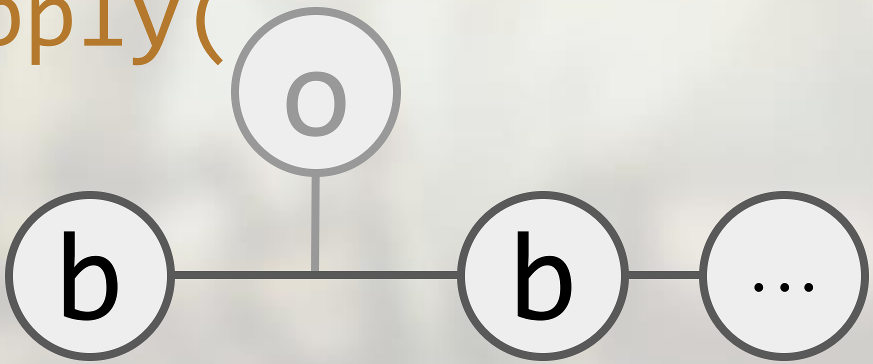


K_b -> A

K_i -> K_o -> A

Infinite Pipes: Merge

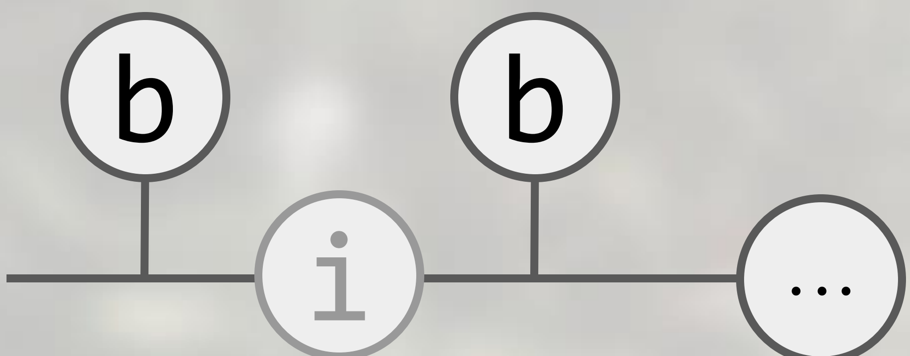
`apply(`



K_i `->` K_o `->` `A`

K_b `->` `A`

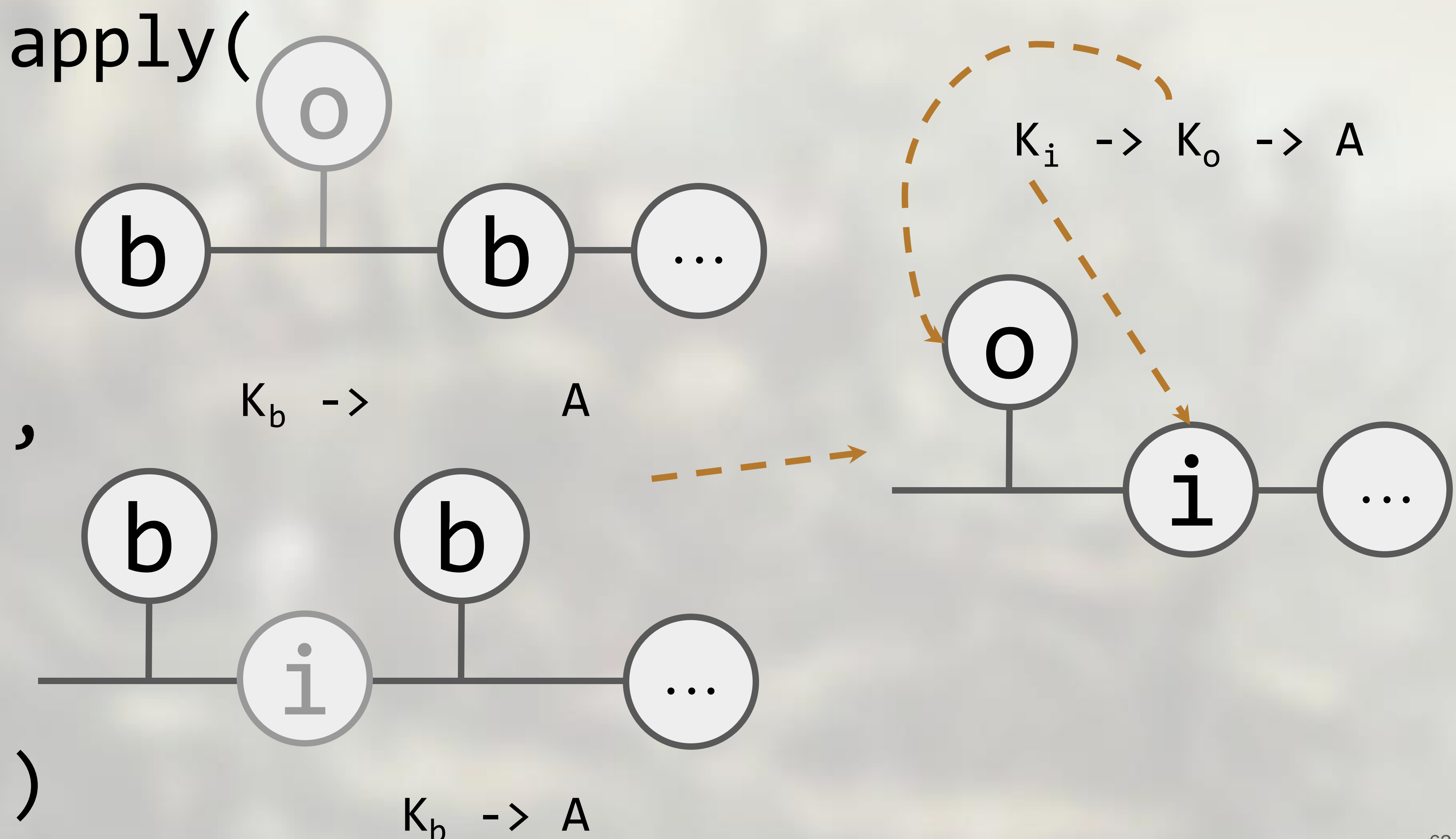
`,`



`)`

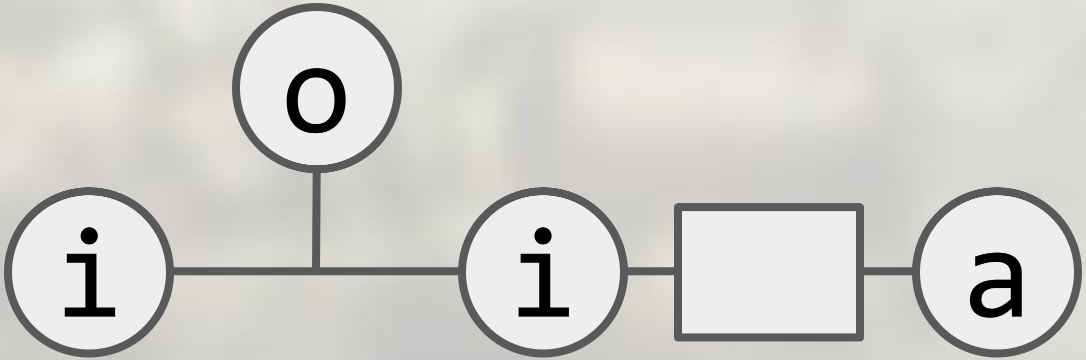
K_b `->` `A`

Infinite Pipes: Merge



Adding Return?

Pipe:



$$: \quad (A \to K_i \to K_o \to R)$$
$$\to K_i \to K_o \to R$$

Adding Return?

Pipe:

$: (A \to K_i \to K_o \to R)$
$\to K_i \to K_o \to R$

Three-Continuation Approach

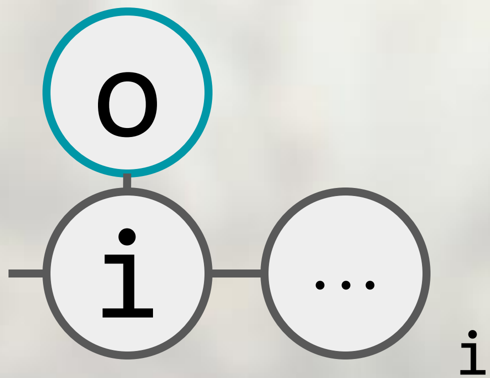
Adding Return?

$: \ (A \to K_b \to K_o \to R)$
$\to K_b \to K_o \to R$

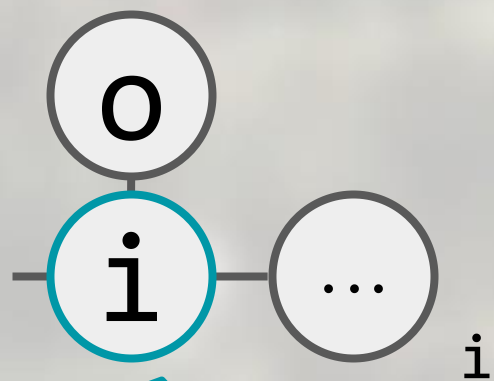
$: \ (A \to K_i \to K_o \to R)$
$\to K_i \to K_o \to R$

$: \ (A \to K_i \to K_b \to R)$
$\to K_i \to K_b \to R$

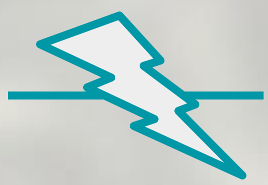
Extended To Bidirectional Pipes



Request



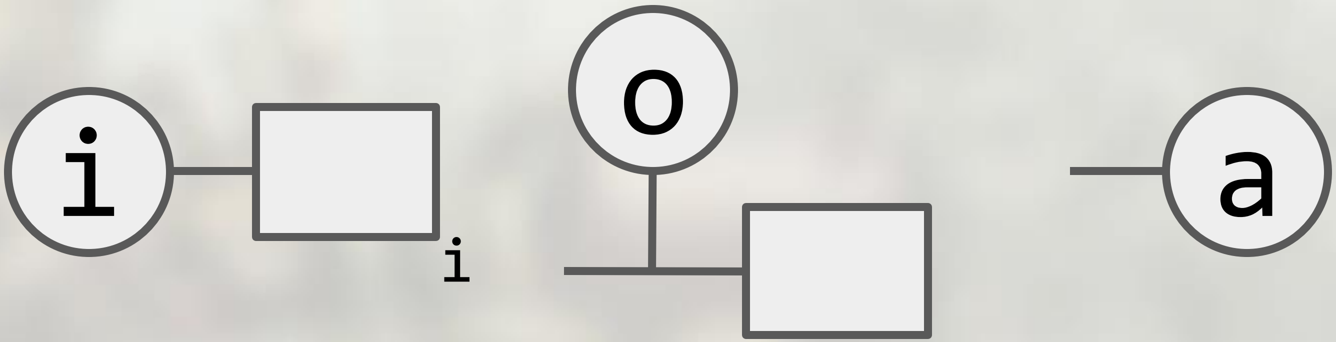
Respond



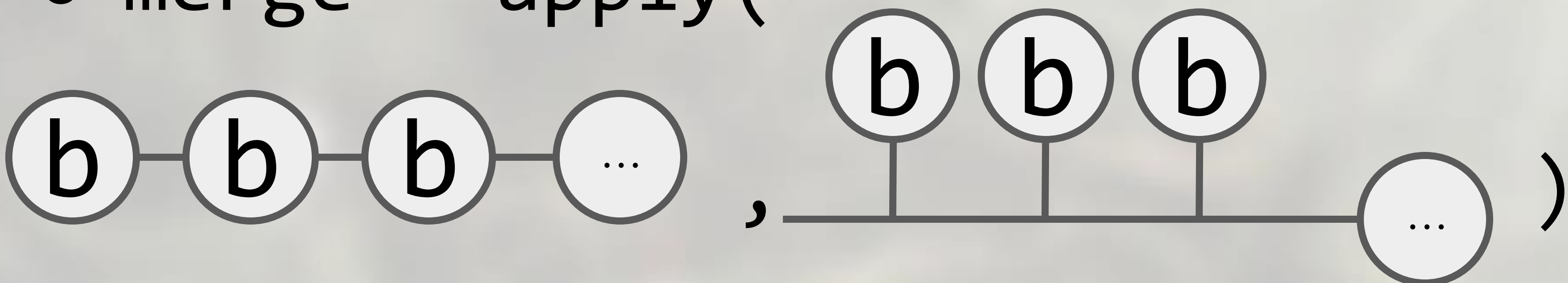
Lift

Summary

- Building Blocks



- `merge = apply(`



ruben.pieters@cs.kuleuven.be
Images source: *Machinarium*