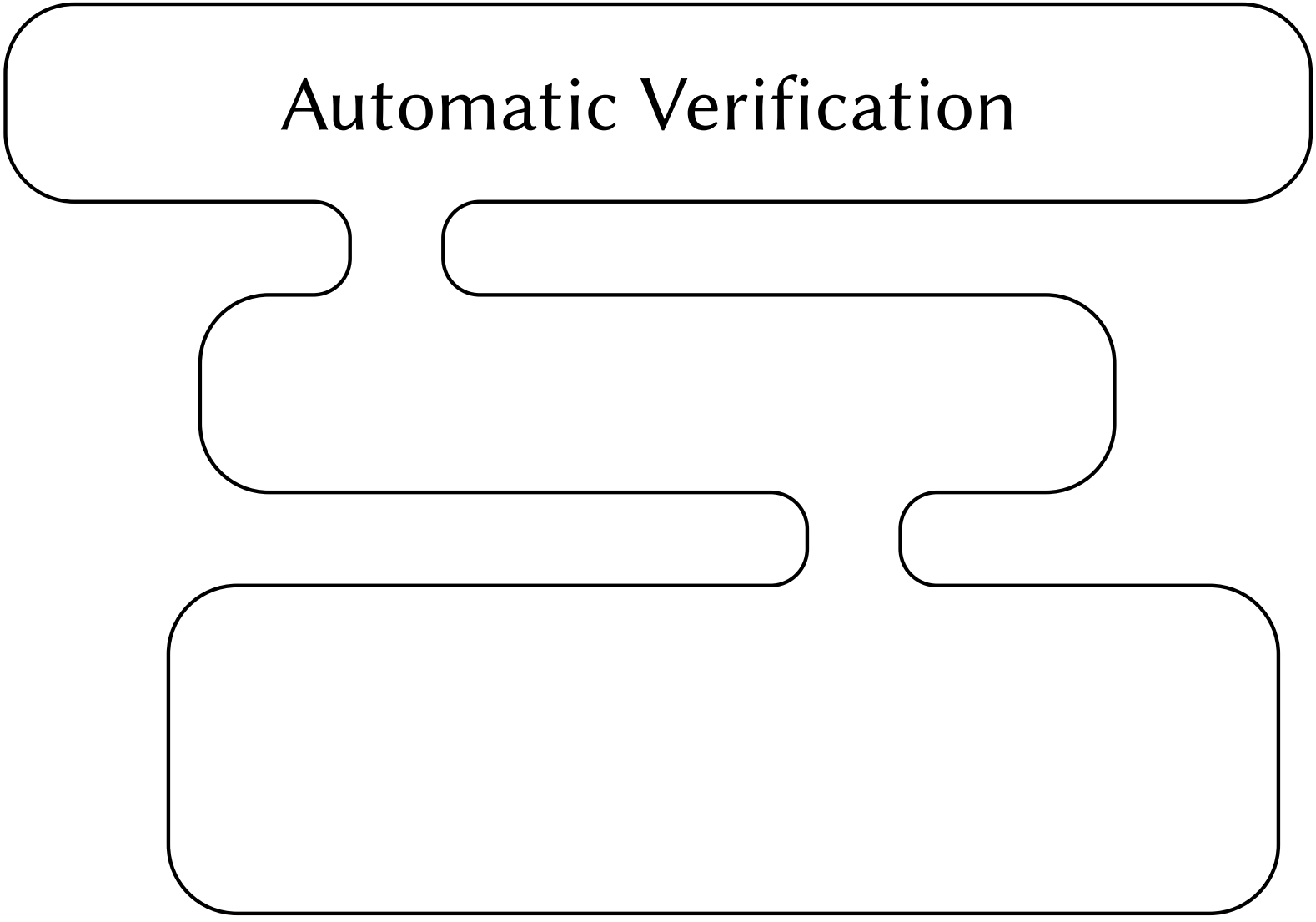
Verification of Erlang-style Concurrency

Emanuele D'Osualdo, Jonathan Kochems and Luke Ong

Department of Computer Science
University of Oxford

11 September 2012

The goal



The goal

Automatic Verification

of

The goal

Automatic Verification

of

Properties

The goal

Automatic Verification

of

Properties

of

The goal

Automatic Verification

of

Properties

of

Concurrent Systems

The goal

Automatic Verification

of

Properties

of

Concurrent Systems

based on Message Passing

The goal

Automatic Verification

of

Properties

of

Concurrent Systems

based on the Actor Model

The goal

Automatic Verification

of

Properties

of

Erlang programs

functional sequential fragment

dynamic process creation

asynchronous message passing

The goal

Effective Sound Approximation

of

Properties

of

Erlang programs

functional sequential fragment

dynamic process creation

asynchronous message passing

The goal

Effective Sound Approximation

of

Reachability

of

Erlang programs

functional sequential fragment

dynamic process creation

asynchronous message passing

Example: Erathostene's sieve

Running Example: a concurrent version of Erathostene's sieve

Inspired by:

Rob Pike.
Concurrency and message passing in Newsqueak.
Google Tech Talks, 2007.

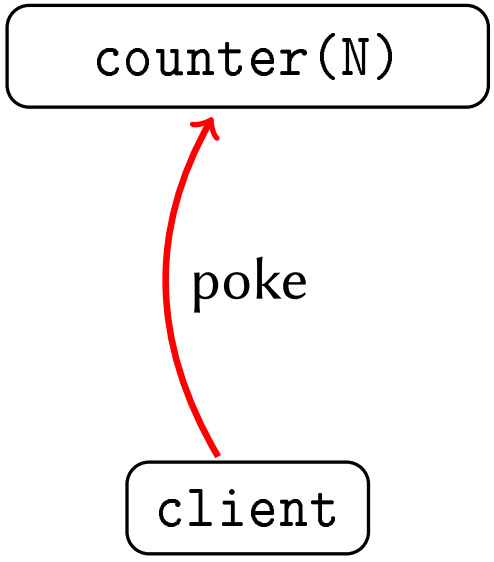
Example: Erathostene's sieve

counter(N)

```
1 counter(N) →
2   receive {poke, From} →
3     From!{ans, N},
4     counter(N+1)
5   end.
```

Example: Erathostene's sieve

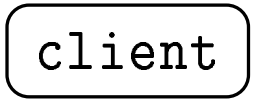

```
1 counter(N) →
2   receive {poke, From} →
3     From!{ans, N},
4     counter(N+1)
5   end.
```



Example: Erathostene's sieve

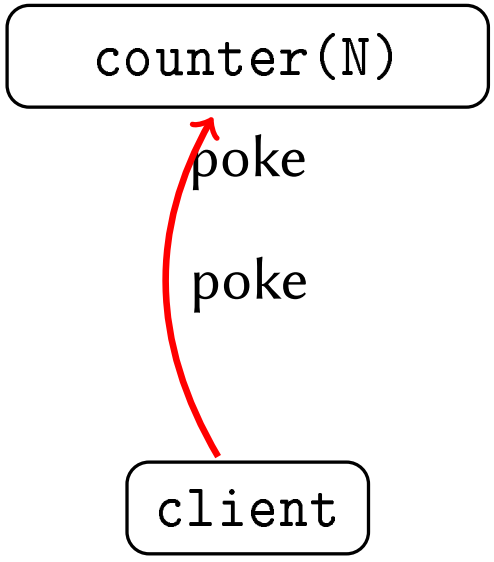

```
1 counter(N) →
2   receive {poke, From} →
3     From!{ans, N},
4     counter(N+1)
5   end.
```

poke



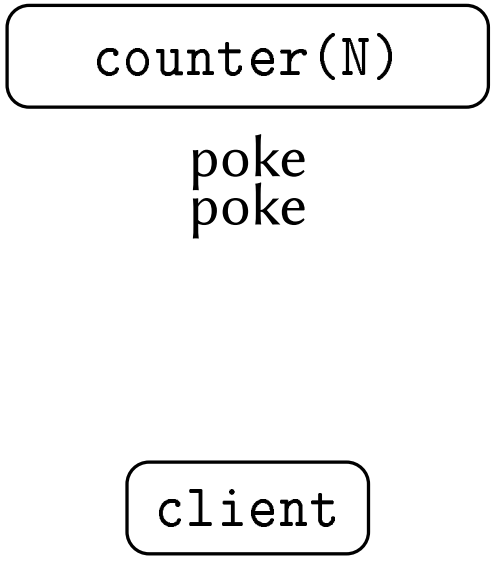
Example: Erathostene's sieve


```
1 counter(N) →
2   receive {poke, From} →
3     From!{ans, N},
4     counter(N+1)
5   end.
```



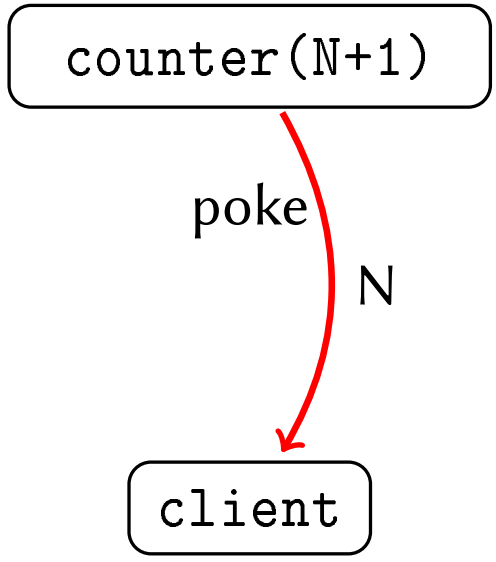
Example: Erathostene's sieve


```
1 counter(N) →
2   receive {poke, From} →
3     From!{ans, N},
4     counter(N+1)
5   end.
```

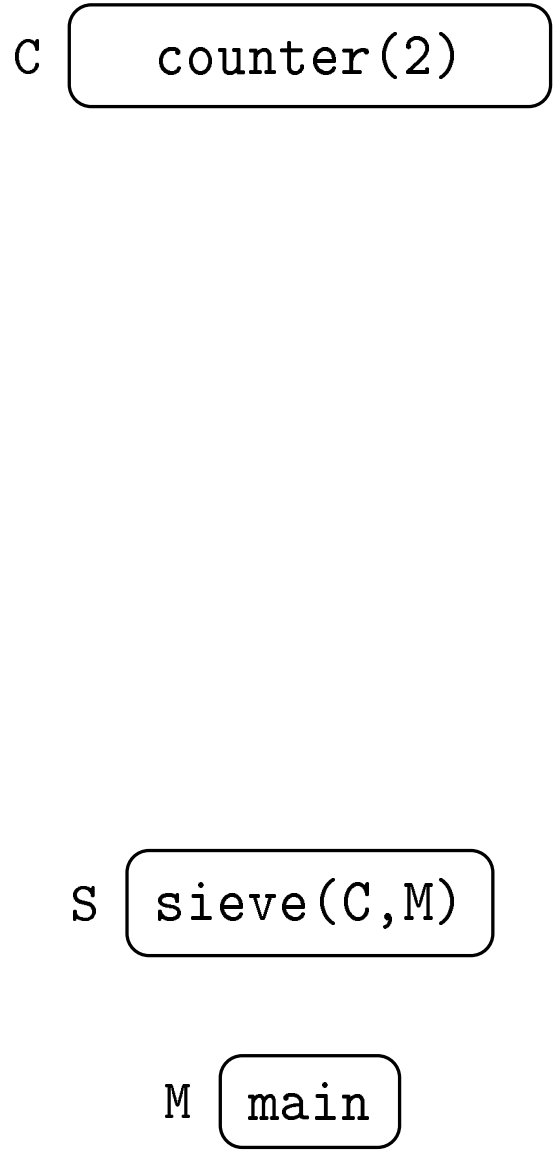


Example: Erathostene's sieve


```
1 counter(N) →
2   receive {poke, From} →
3     From!{ans, N},
4     counter(N+1)
5   end.
```

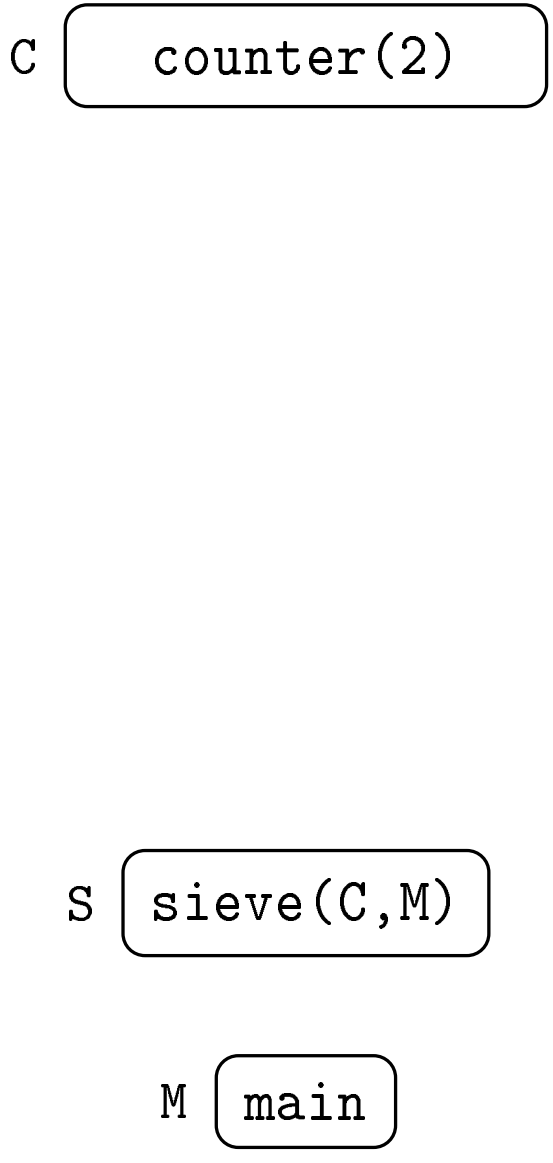


Example: Erathostene's sieve




```
1 main() →
2   M = self(),
3   C = spawn(counter,[2]),
4   spawn(sieve, [C,M]).
```

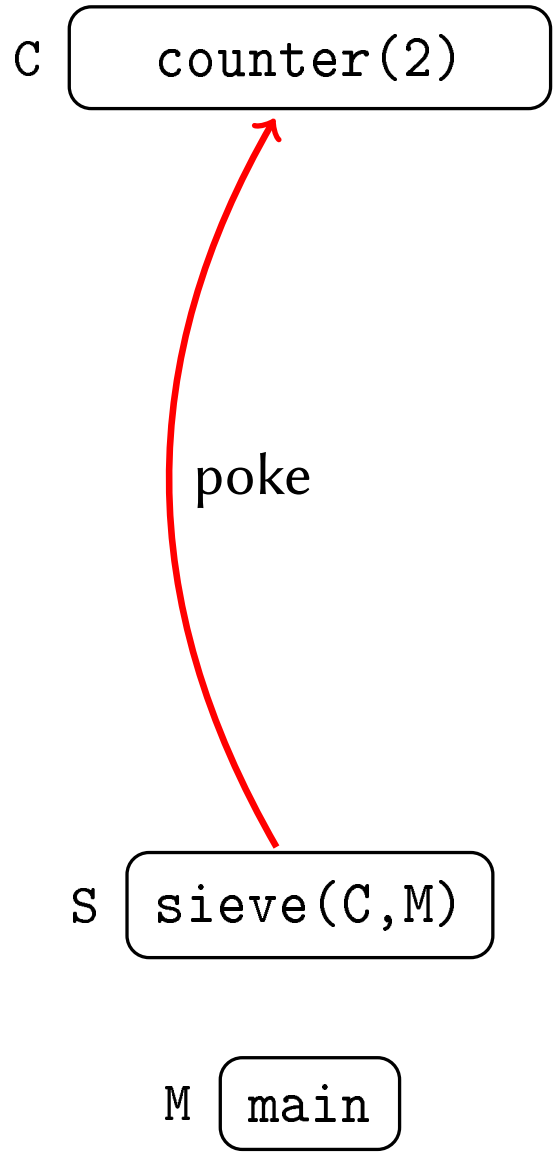
Example: Erathostene's sieve



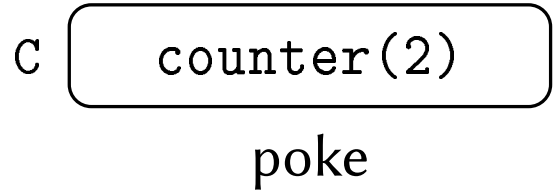

```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```

Example: Erathostene's sieve

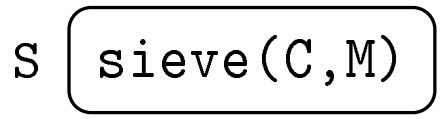

```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```



Example: Erathostene's sieve

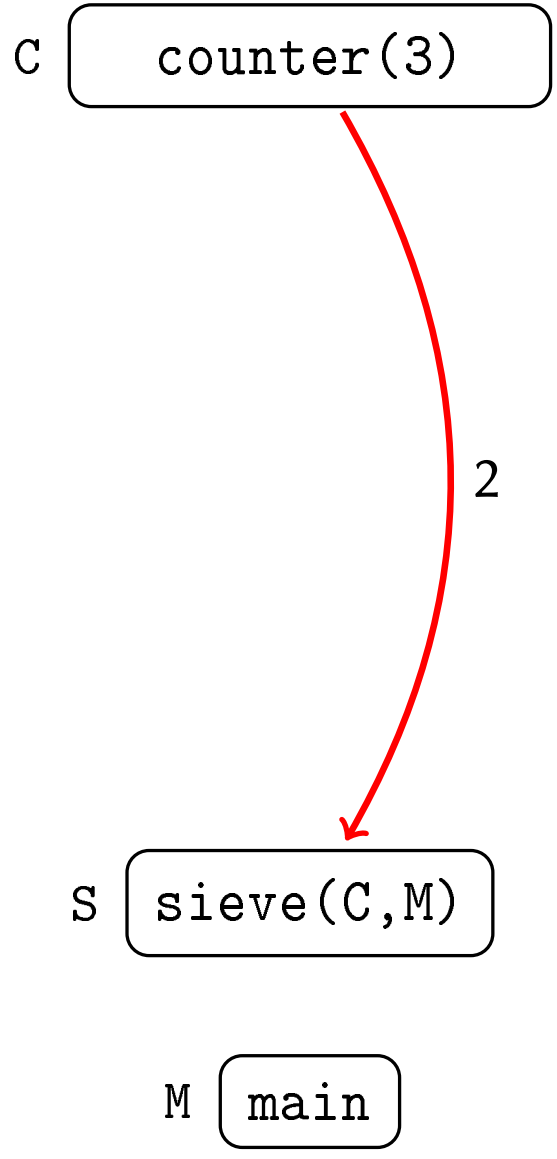



```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```



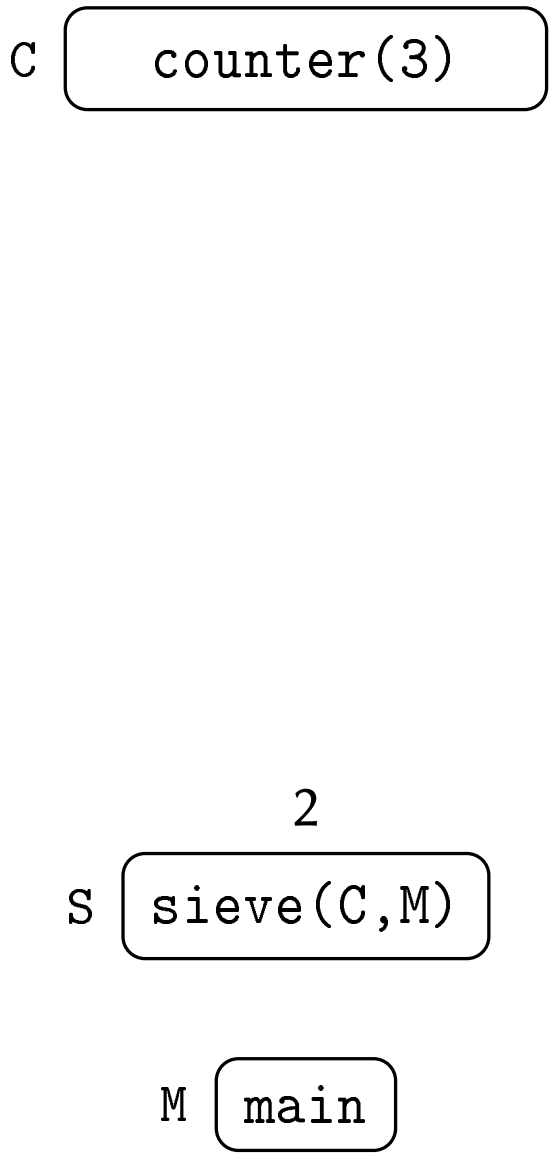
Example: Erathostene's sieve


```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```

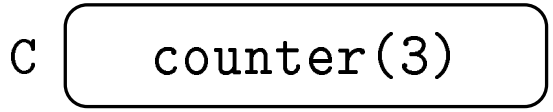


Example: Erathostene's sieve

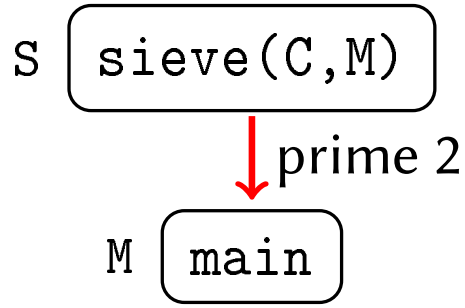

```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```



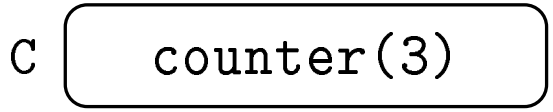
Example: Erathostene's sieve



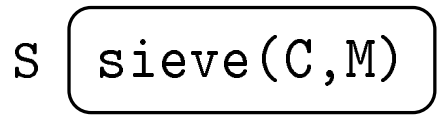

```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```

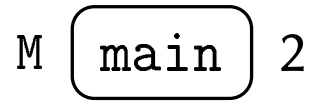


Example: Erathostene's sieve

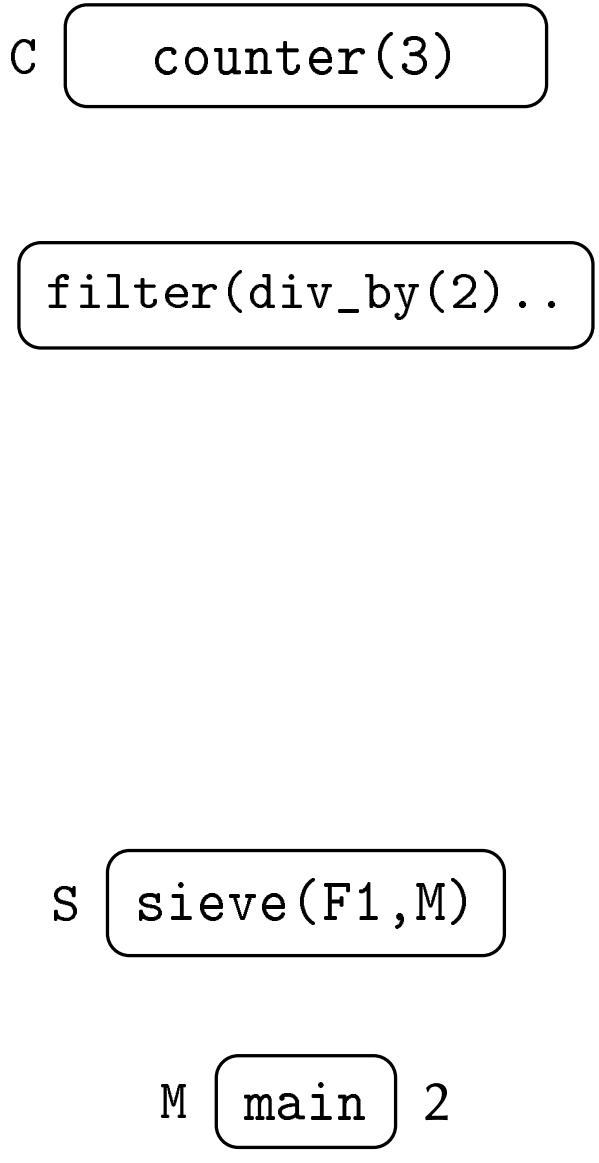



```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```





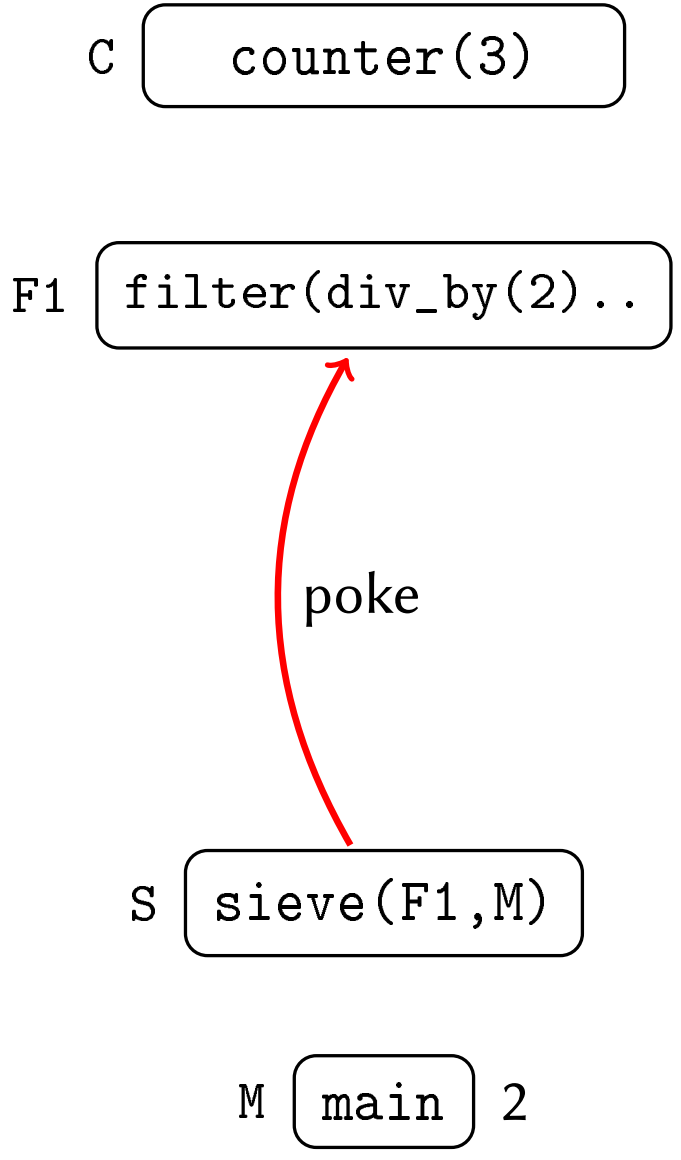
Example: Erathostene's sieve




```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```

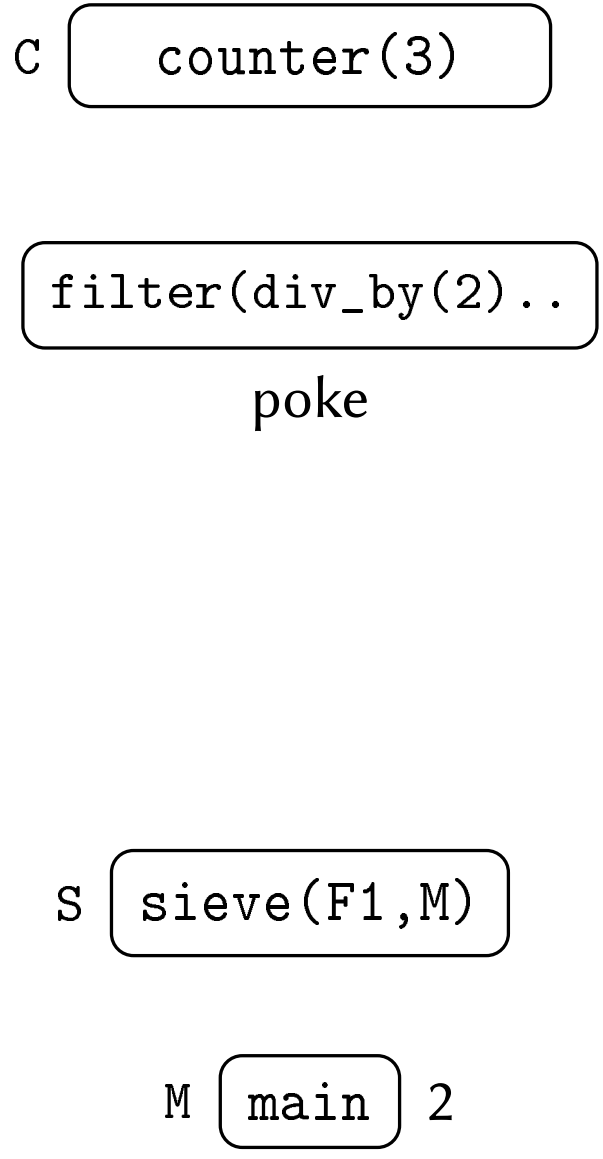
Example: Erathostene's sieve


```
1 sieve(In, Out) →
2   In!{poke, self()},
3   receive {ans,X} →
4     Out!{prime,X},
5     F = spawn(fun()→
6           filter(div_by(X), In)
7         end),
8     sieve(F,Out)
9   end.
```



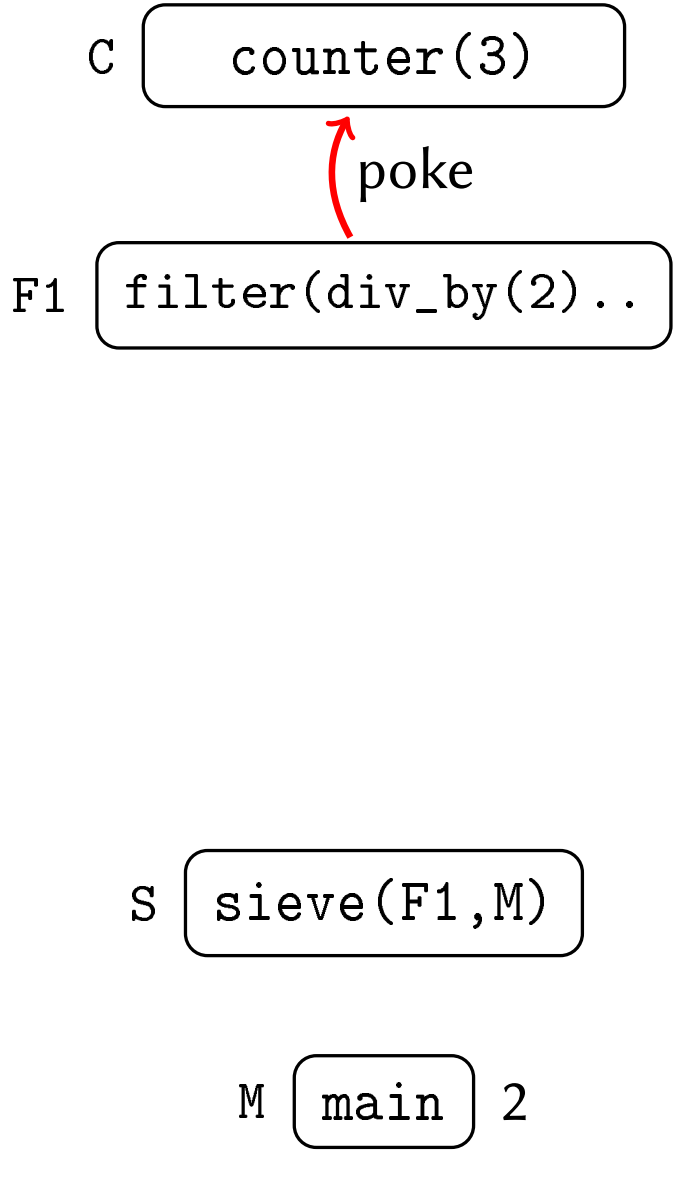
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



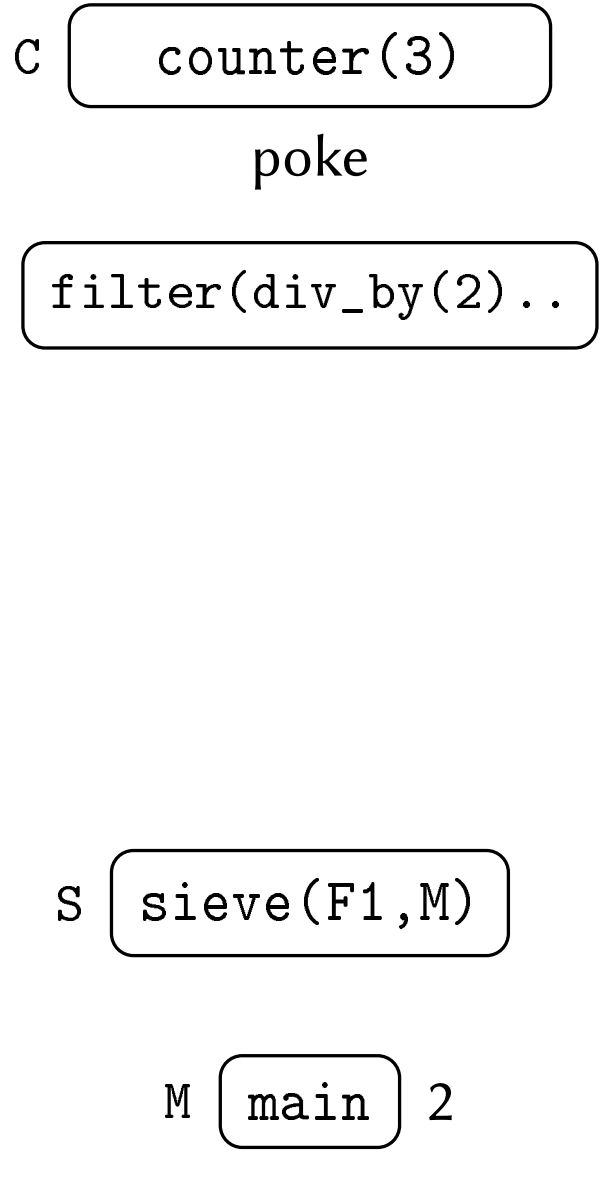
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



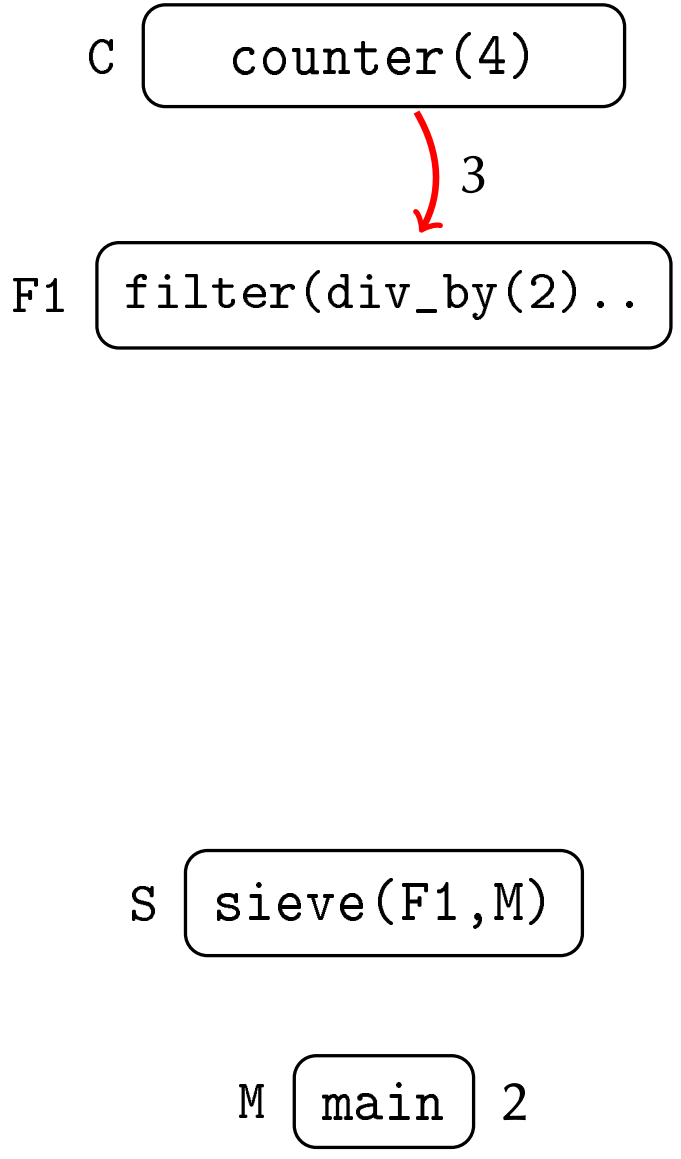
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



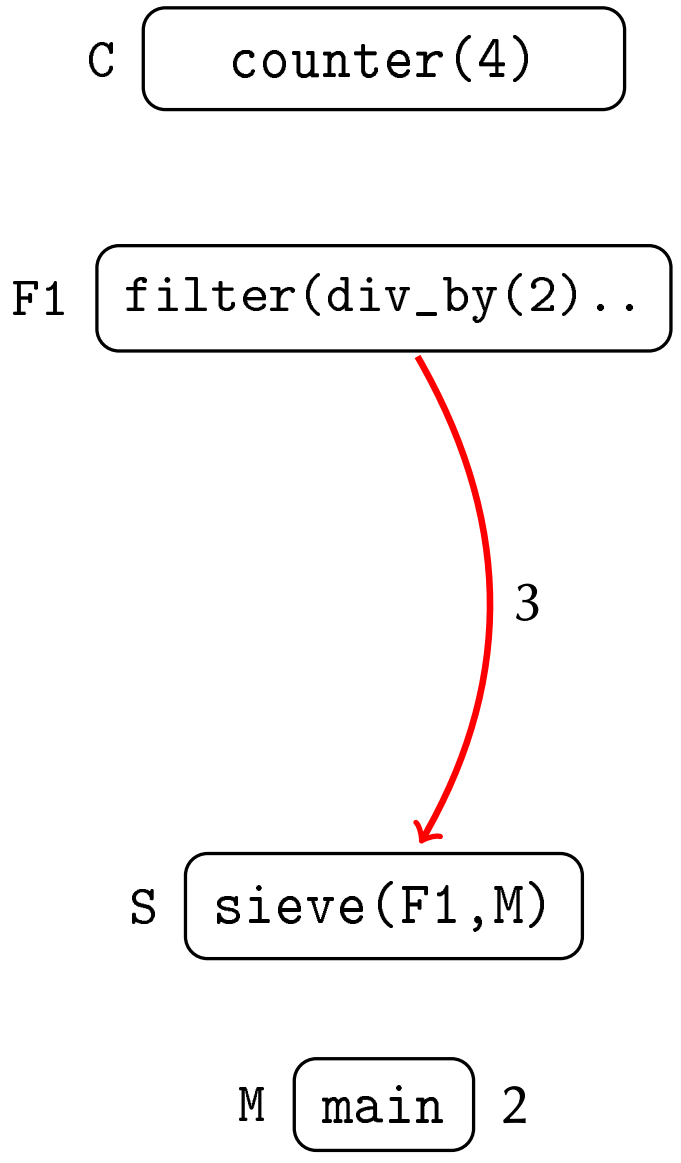
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



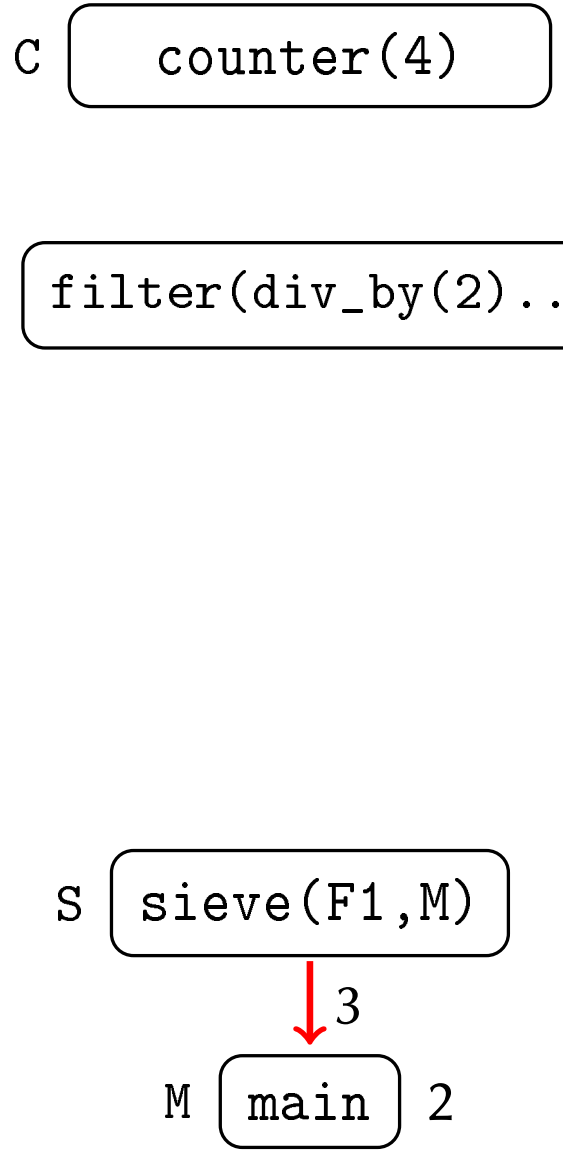
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



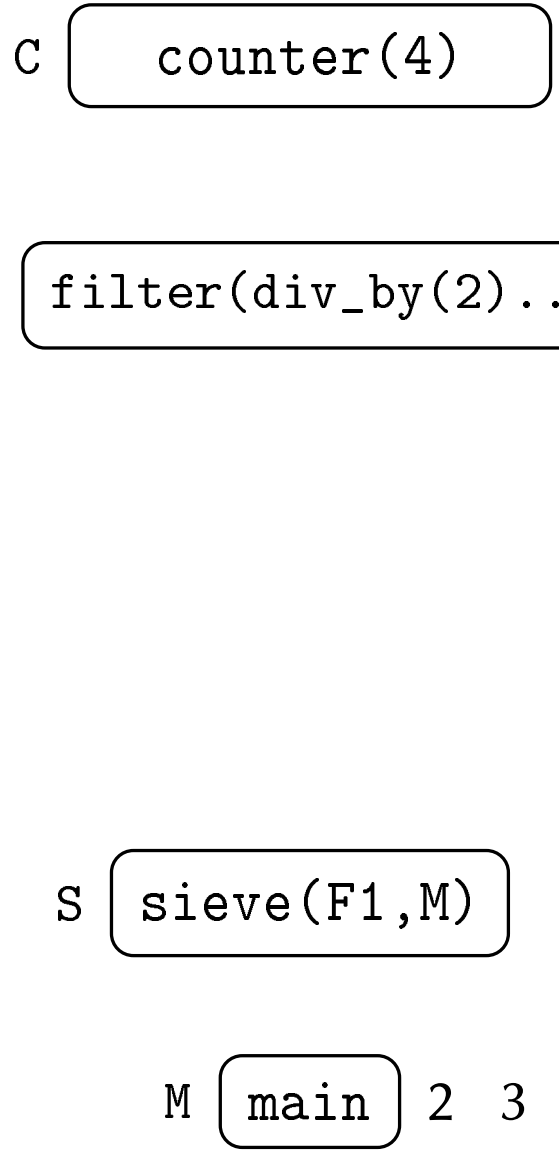
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



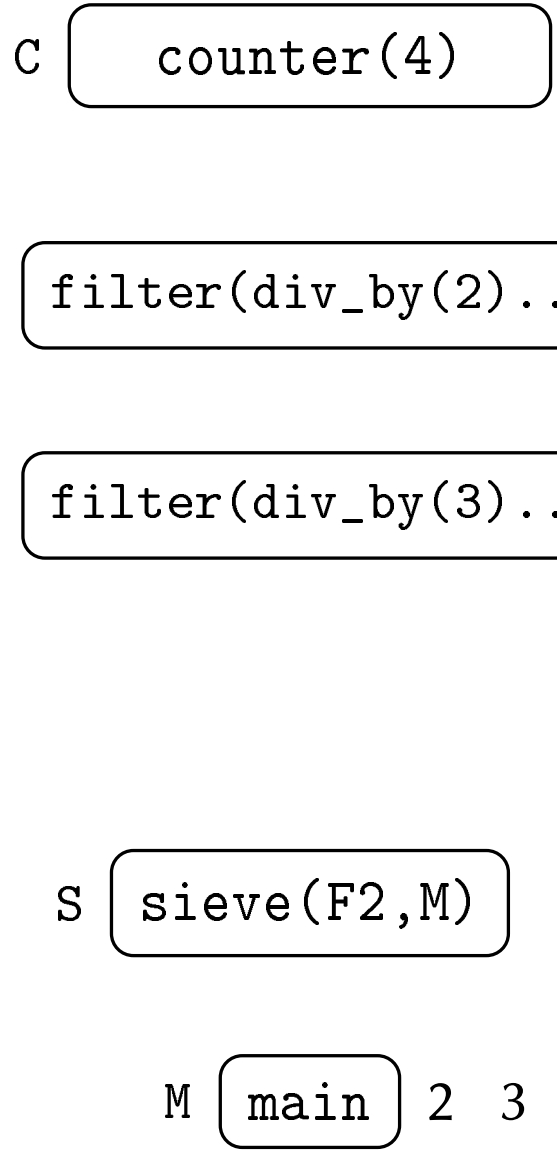
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



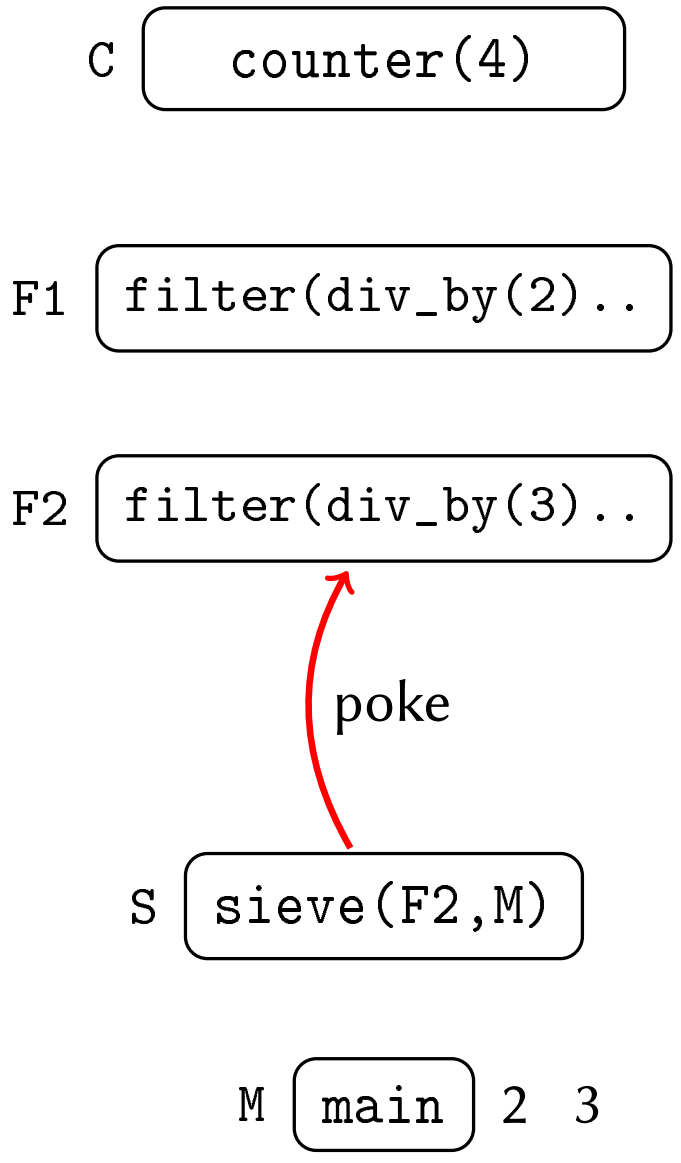
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



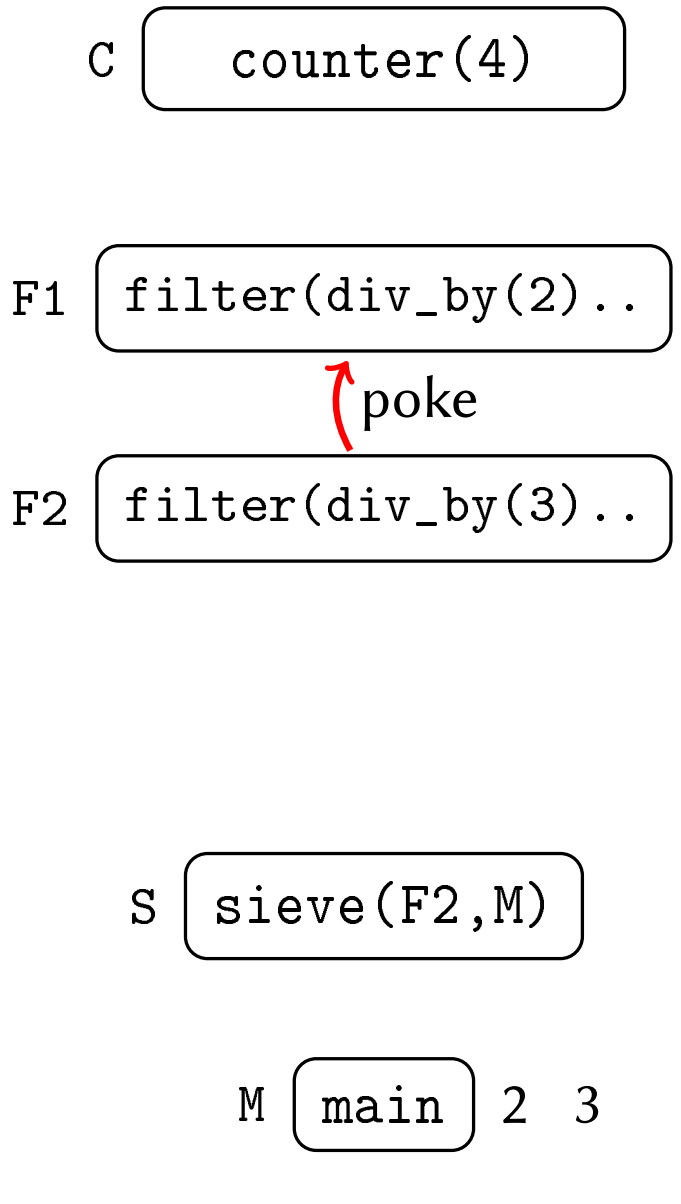
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



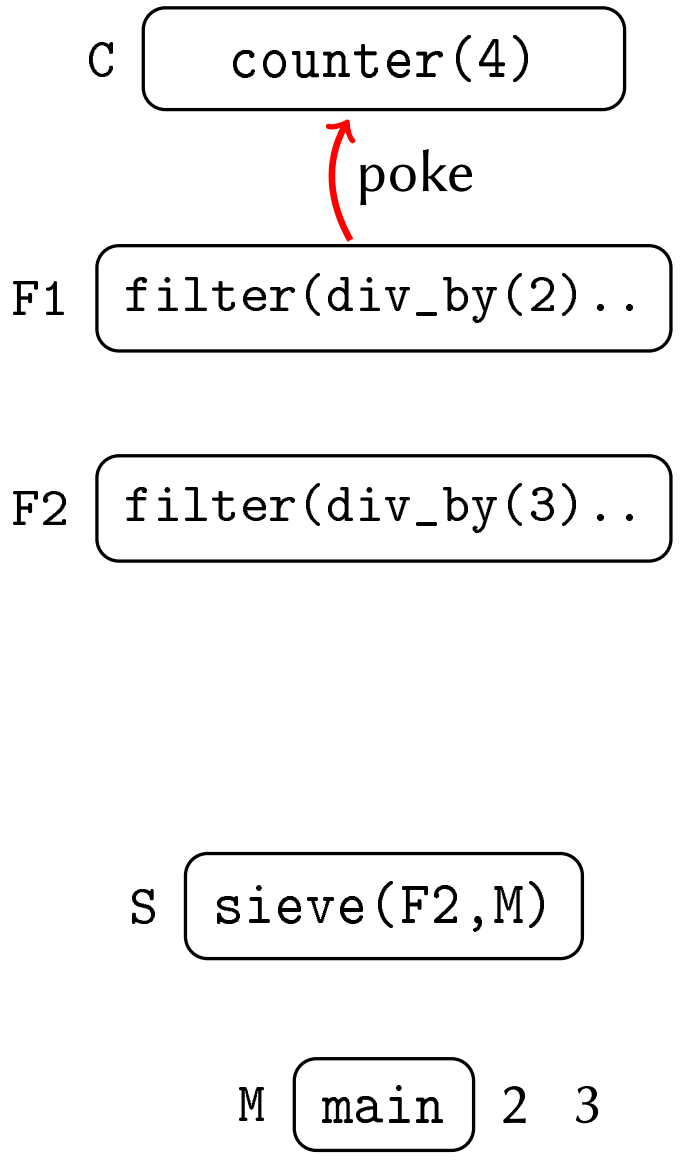
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



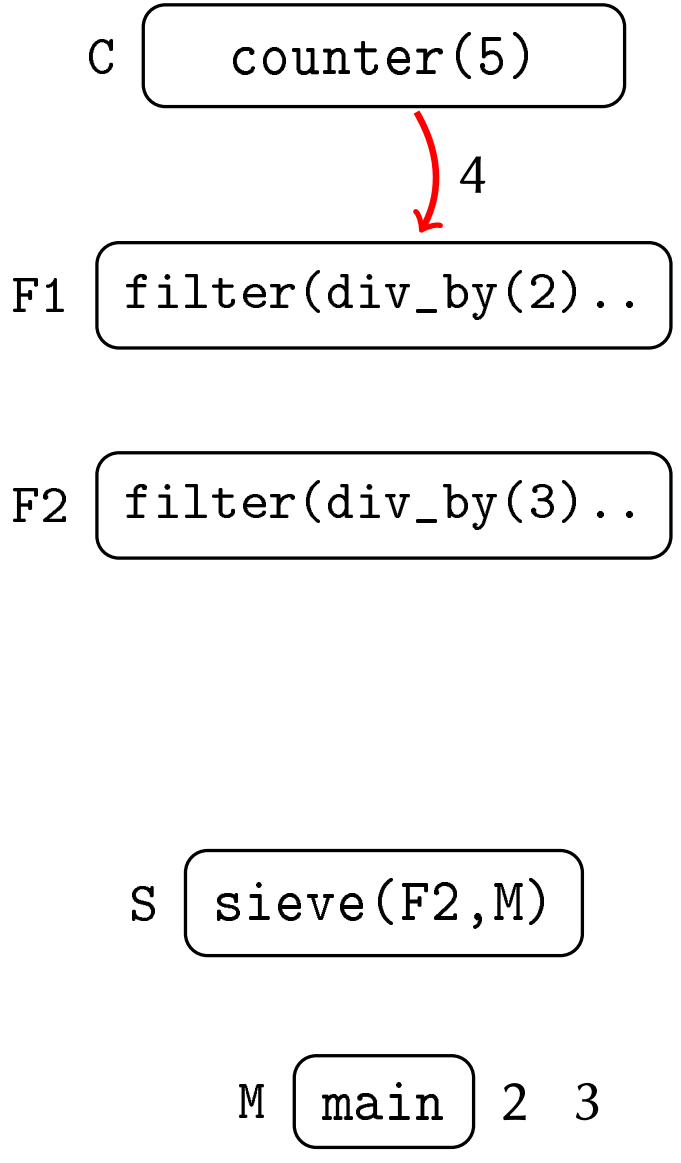
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



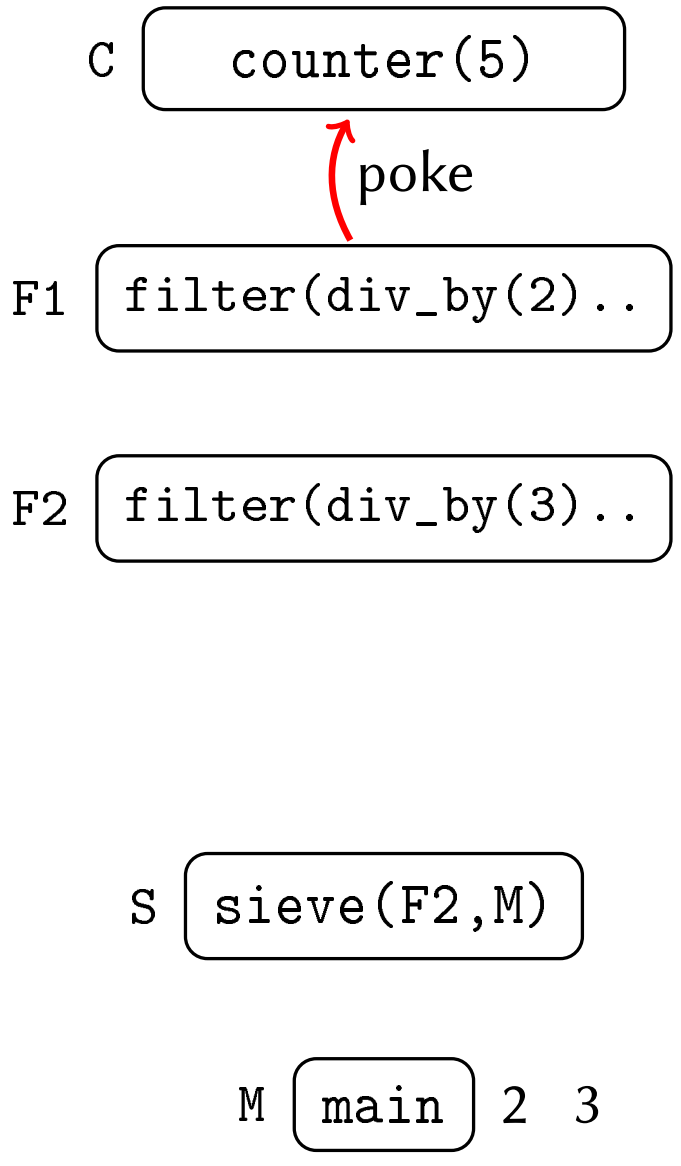
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



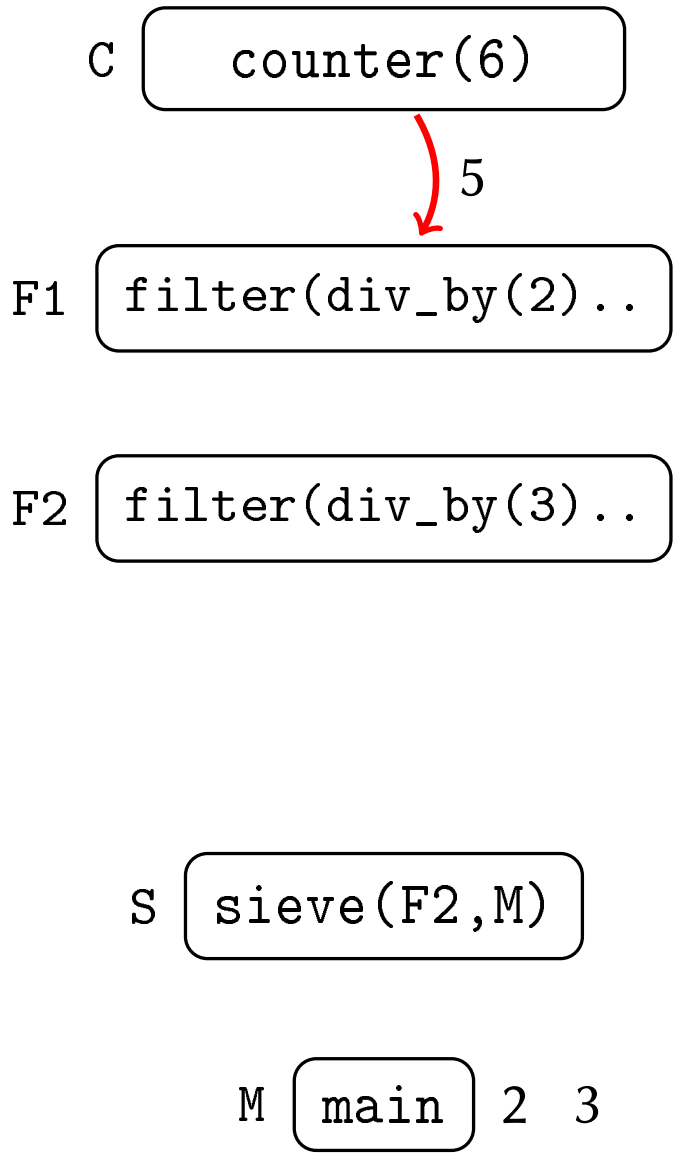
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



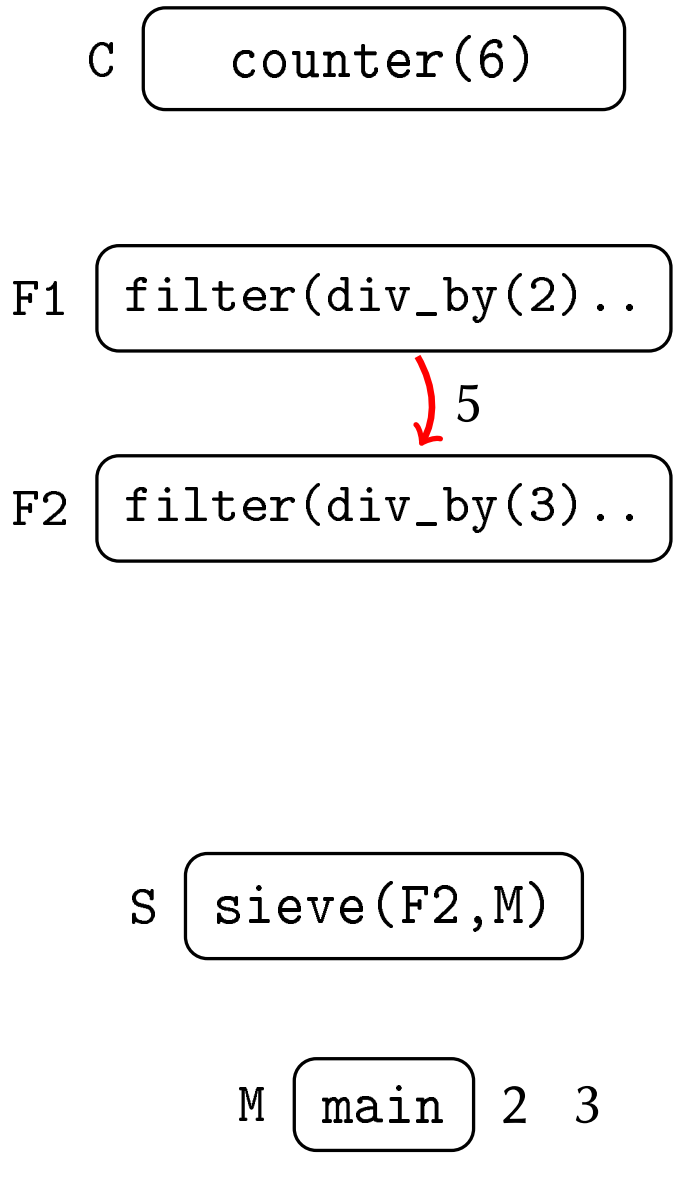
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



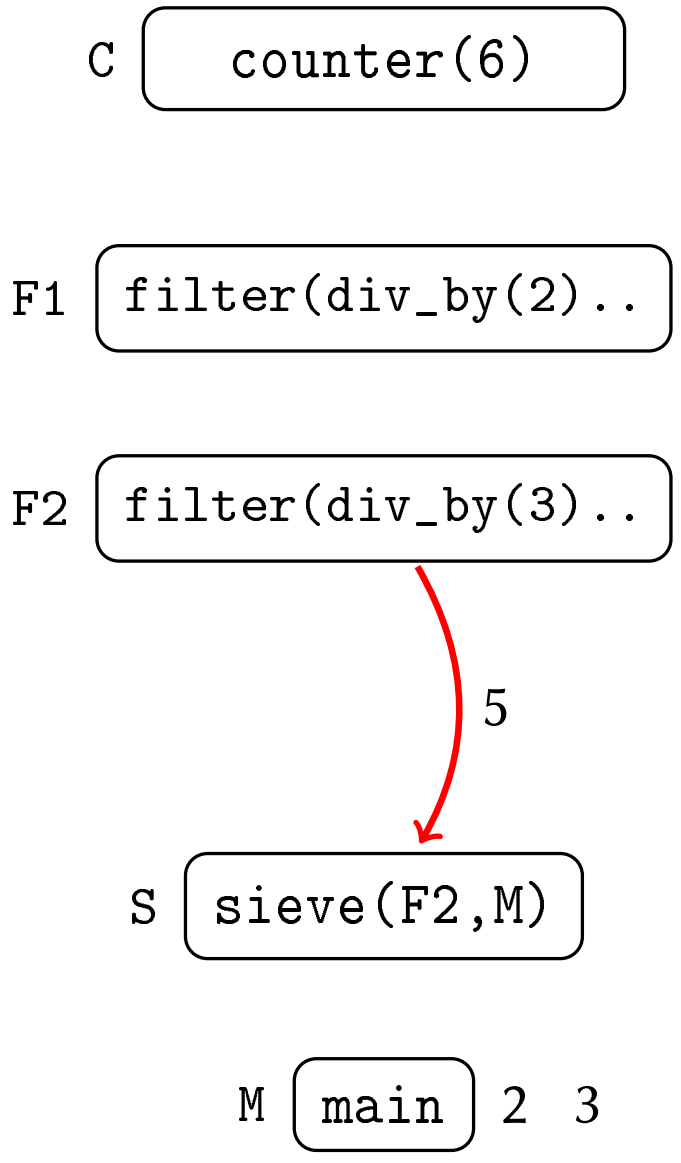
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



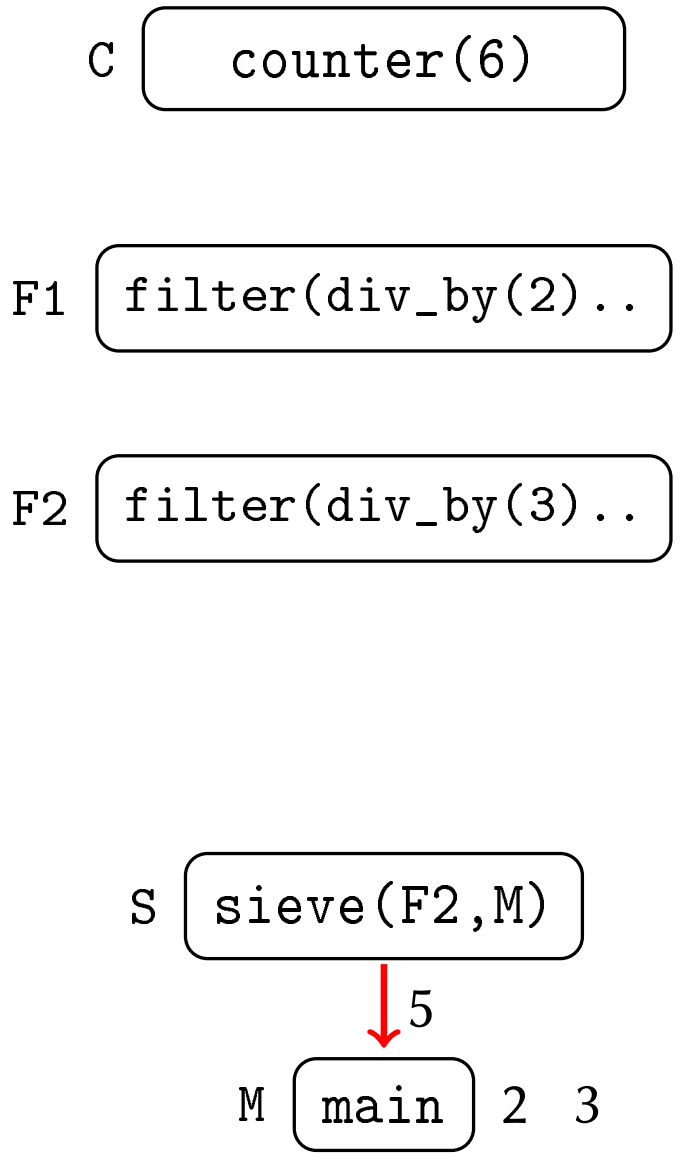
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



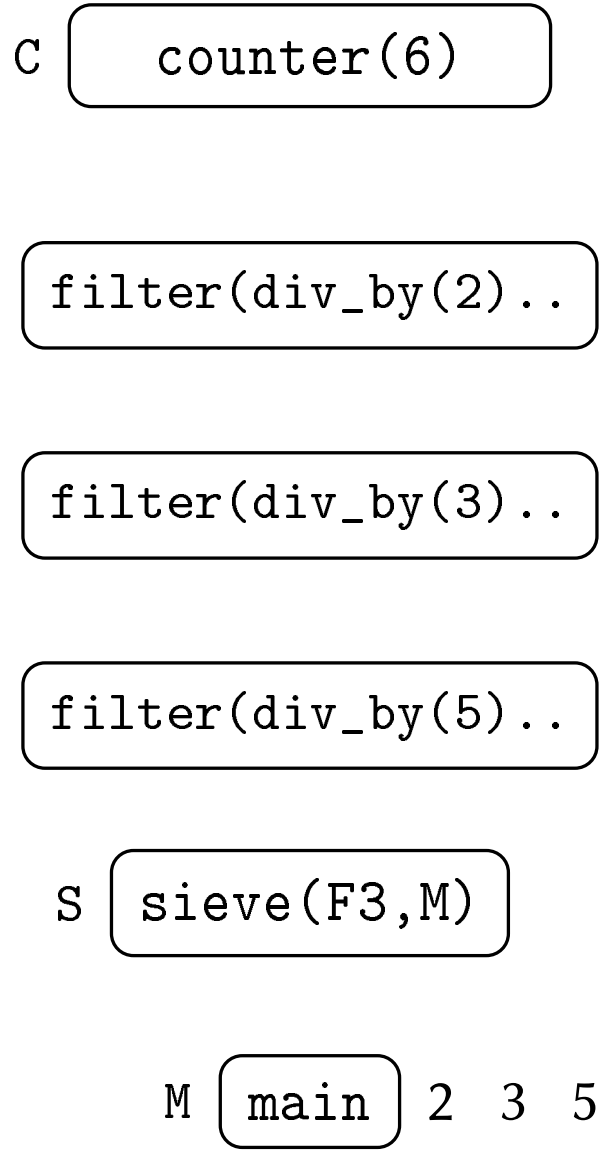
Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



Example: Erathostene's sieve


```
1  filter(Test, In) →
2    receive {poke, From} →
3      filter(Test, In, From)
4    end.
5
6  filter(Test, In, Out) →
7    In!{poke, self()},
8    receive {ans,Y} →
9      case Test(Y) of
10       false →
11         Out!{ans,Y},
12         filter(Test, In);
13       true →
14         filter(Test, In, Out)
15     end
16   end.
```



Erlang

Erlang:

- Pure functional sequential fragment
- Call-by-value
- Dynamically typed
- Higher Order
- Process creation: `spawn` / `self`
- Message passing: `receive` / send `P ! Msg`
- Send is asynchronous, receive is blocking

Jim Larson.
Erlang for concurrent programming.
Communications of the ACM, 2009.

Related Work

Bug-finding tools:

Related Work

Bug-finding tools:

- **Dialyzer** (Lindahl&Sagonas)
  based on control-flow-analysis, *success types* and detection of wrong use of built-ins

Related Work

Bug-finding tools:

- **Dialyzer** (Lindahl&Sagonas)
  based on control-flow-analysis, *success types* and detection of wrong use of built-ins

- **QuickCheck** (Arts et al.) / **PropEr** (Sagonas' group)
  property-based testing

Related Work

Bug-finding tools:

- **Dialyzer** (Lindahl&Sagonas)
  based on control-flow-analysis, *success types* and detection of wrong use of built-ins

- **QuickCheck** (Arts et al.) / **PropEr** (Sagonas' group)
  property-based testing

Related Work

Verification tools:

Related Work

Verification tools:

- **McErlang** (Fredlund&Svensson)
  instrumented custom runtime which exhaustively explores (on-the-fly) all the execution paths.
  Parametrised by user-provided abstractions

Related Work

Verification tools:

- **McErlang** (Fredlund&Svensson)
  instrumented custom runtime which exhaustively explores (on-the-fly) all the execution paths.
  Parametrised by user-provided abstractions
  Termination and soundness depend entirely on user params

Related Work

Verification tools:

- **McErlang** (Fredlund&Svensson)
  instrumented custom runtime which exhaustively explores (on-the-fly) all the execution paths.
  Parametrised by user-provided abstractions
  Termination and soundness depend entirely on user params

- **Abstract Model Checking** (Huch)
  reduction of the operational semantics to a sound *finite* transition system via user-provided data-abstractions

Related Work

Verification tools:

- **McErlang** (Fredlund&Svensson)
  instrumented custom runtime which exhaustively explores (on-the-fly) all the execution paths.
  Parametrised by user-provided abstractions
  Termination and soundness depend entirely on user params

- **Abstract Model Checking** (Huch)
  reduction of the operational semantics to a sound *finite* transition system via user-provided data-abstractions
  Applies to a very restricted fragment of Erlang

Erlang

Goals:

SAFETY

PRECISION

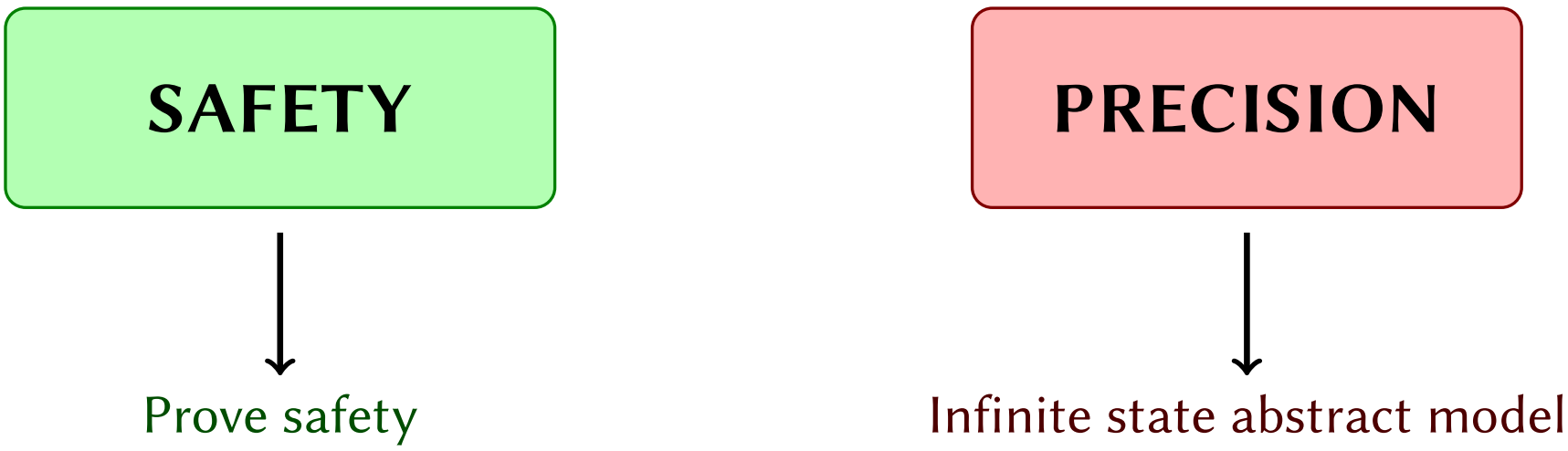
Goals:

SAFETY

Prove safety

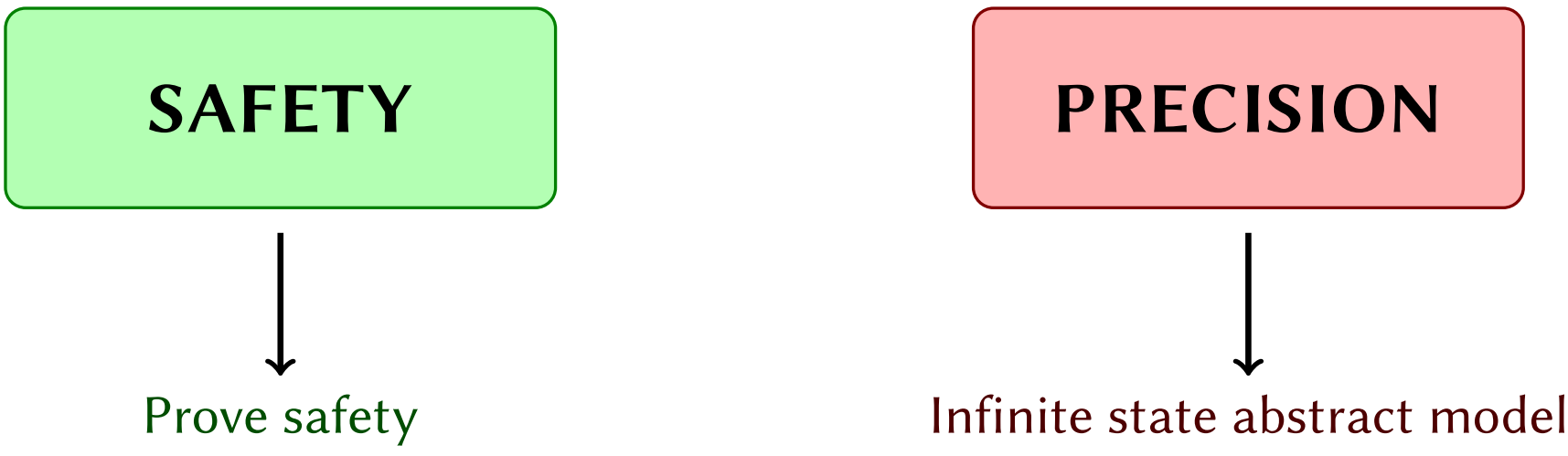
PRECISION

Goals:


SAFETY
PRECISION
Prove safety
Infinite state abstract model

Erlang

Goals:



Must abstract to be automatic!

Why is it difficult to verify?

Sources of infinity in the state space:

- recursive function definitions
- higher order
- infinite domains of values
- message space is infinite
- unbounded dynamic processes creation
- mailboxes have unbounded capacity

Why is it difficult to verify?

Sources of infinity in the state space:

- recursive function definitions **CFA-like abs**
- higher order **CFA-like abs**
- infinite domains of values **CFA-like abs**
- message space is infinite
- unbounded dynamic processes creation
- mailboxes have unbounded capacity

Why is it difficult to verify?

Sources of infinity in the state space:

- recursive function definitions **CFA-like abs**
- higher order **CFA-like abs**
- infinite domains of values **CFA-like abs**
- message space is infinite **CFA-like abs**
- unbounded dynamic processes creation **CFA-like abs**
- mailboxes have unbounded capacity **CFA-like abs**

Using methodology from

Van Horn & Might.
Abstracting Abstract Machines.
ICFP, 2010.

Why is it difficult to verify?

Sources of infinity in the state space:

- recursive function definitions — **CFA-like abs**
- higher order — **CFA-like abs**
- infinite domains of values — **CFA-like abs**
- message space is infinite — **finite abs**
- unbounded dynamic processes creation — **counter abs**
- mailboxes have unbounded capacity — **counter abs**

Actor Communicating Systems

We define an abstract model:

- finite set of *control states* q
- finite set of *messages* $\mathtt{m}$
- finite set of *pid-classes* ι
- finite set of rules:

$$\textbf{Seq. red.} \qquad \iota : q \xrightarrow{\tau} q'$$

$$\textbf{Send} \qquad \iota : q \xrightarrow{\iota'!\mathtt{m}} q'$$

$$\textbf{Receive} \qquad \iota : q \xrightarrow{?\mathtt{m}} q'$$

$$\textbf{Spawn} \qquad \iota : q \xrightarrow{\nu\iota'.q'} q''$$

Actor Communicating Systems

We define an abstract model:

- finite set of *control states* q
- finite set of *messages* $\mathtt{m}$
- finite set of *pid-classes* ι
- finite set of rules:

$$\textbf{Seq. red.} \qquad \iota : q \xrightarrow{\tau} q'$$

$$\textbf{Send} \qquad \iota : q \xrightarrow{\iota'!\mathtt{m}} q'$$

$$\textbf{Receive} \qquad \iota : q \xrightarrow{?\mathtt{m}} q'$$

$$\textbf{Spawn} \qquad \iota : q \xrightarrow{\nu\iota'.q'} q''$$

They are equivalent to *Vector Addition Systems* (VAS)

Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

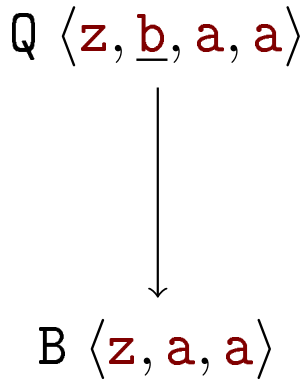
Q ⟨z, b, a, a⟩

B ⟨z, a, a⟩

Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```



Finite State control + FIFFO queue is Turing Powerful

Erlang and ACS mailboxes

Erlang

```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

ACS

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{a}} \mathtt{A}$$

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{b}} \mathtt{B}$$

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{c}} \mathtt{C}$$

$$\mathtt{Q}\ \langle \mathtt{z}, \underline{\mathtt{b}}, \mathtt{a}, \mathtt{a} \rangle$$

$$\downarrow$$

$$\mathtt{B}\ \langle \mathtt{z}, \mathtt{a}, \mathtt{a} \rangle$$

Erlang and ACS mailboxes

Erlang

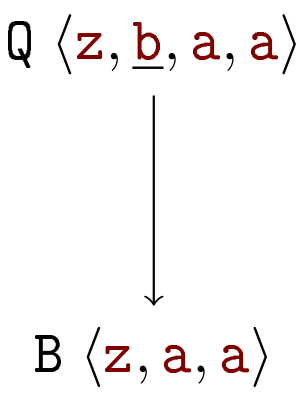
```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

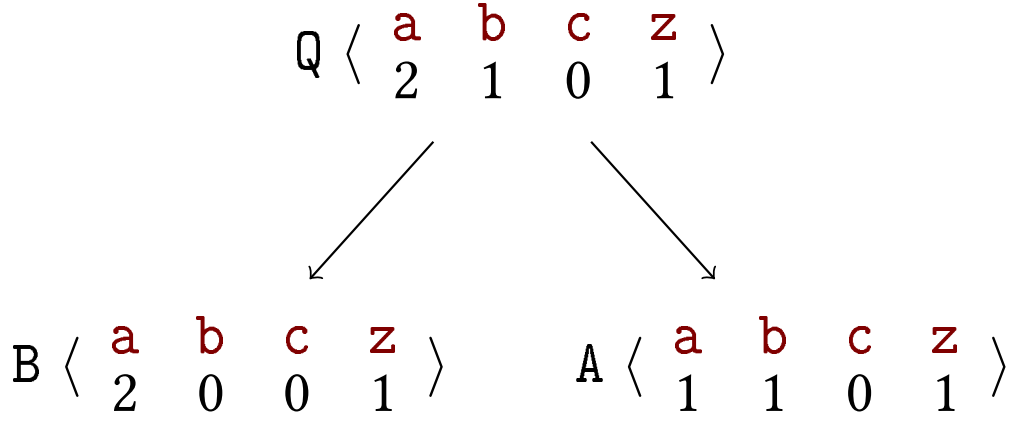
ACS

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{a}} \mathtt{A}$$

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{b}} \mathtt{B}$$

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{c}} \mathtt{C}$$





Erlang and ACS mailboxes

Erlang

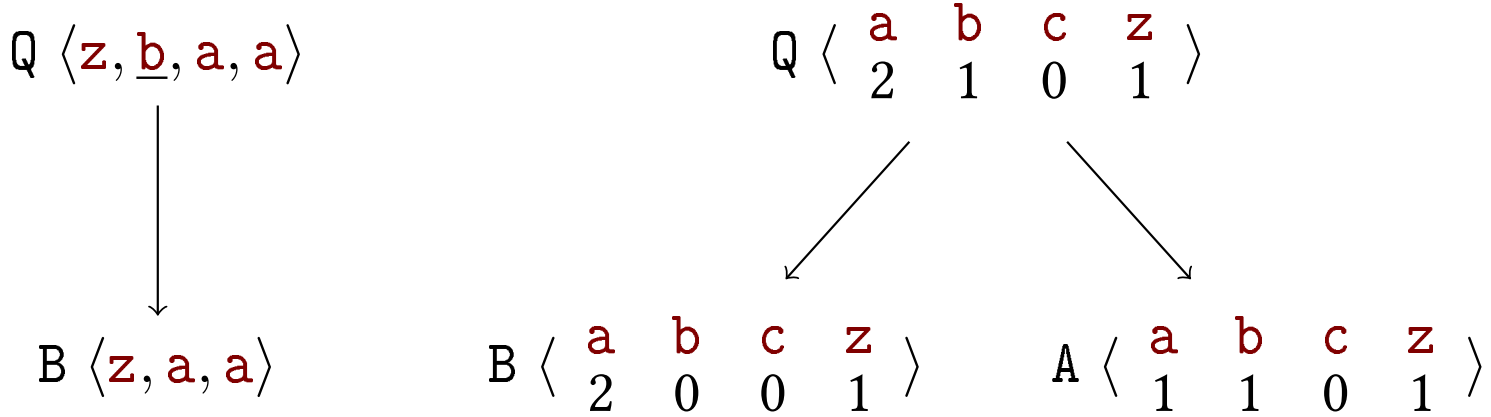
```
Q →
    receive
        a → A;
        b → B;
        c → C
    end.
```

ACS

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{a}} \mathtt{A}$$

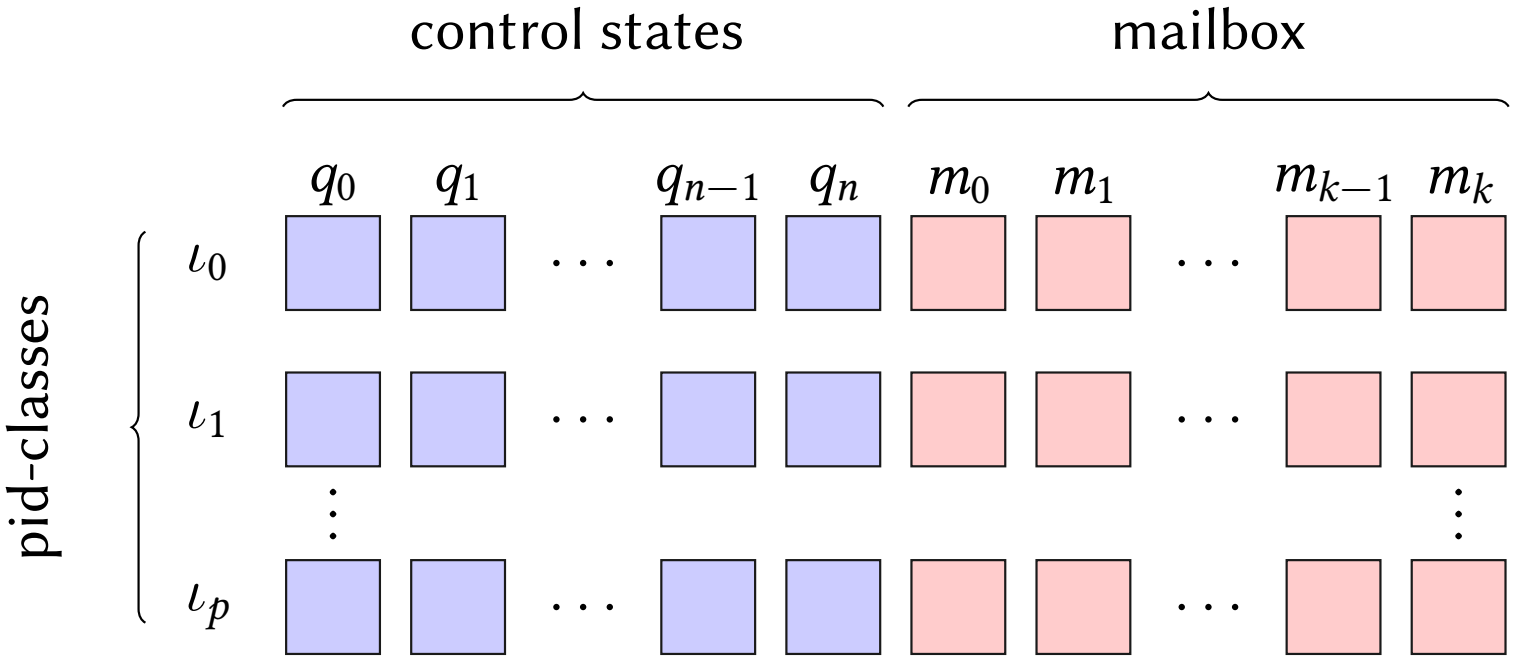
$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{b}} \mathtt{B}$$

$$\iota : \mathtt{Q} \xrightarrow{?\mathtt{c}} \mathtt{C}$$

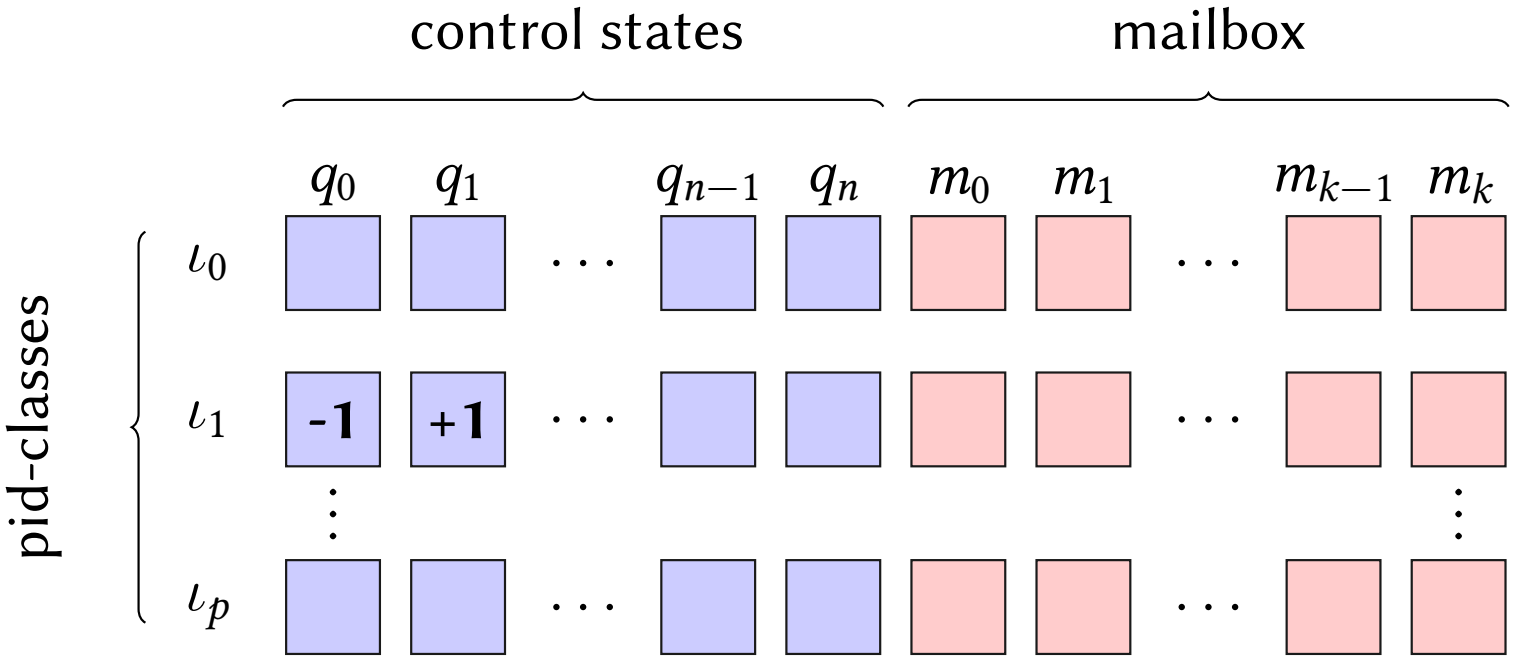


ACS mailboxes over-approximate Erlang ones

Underlying VAS

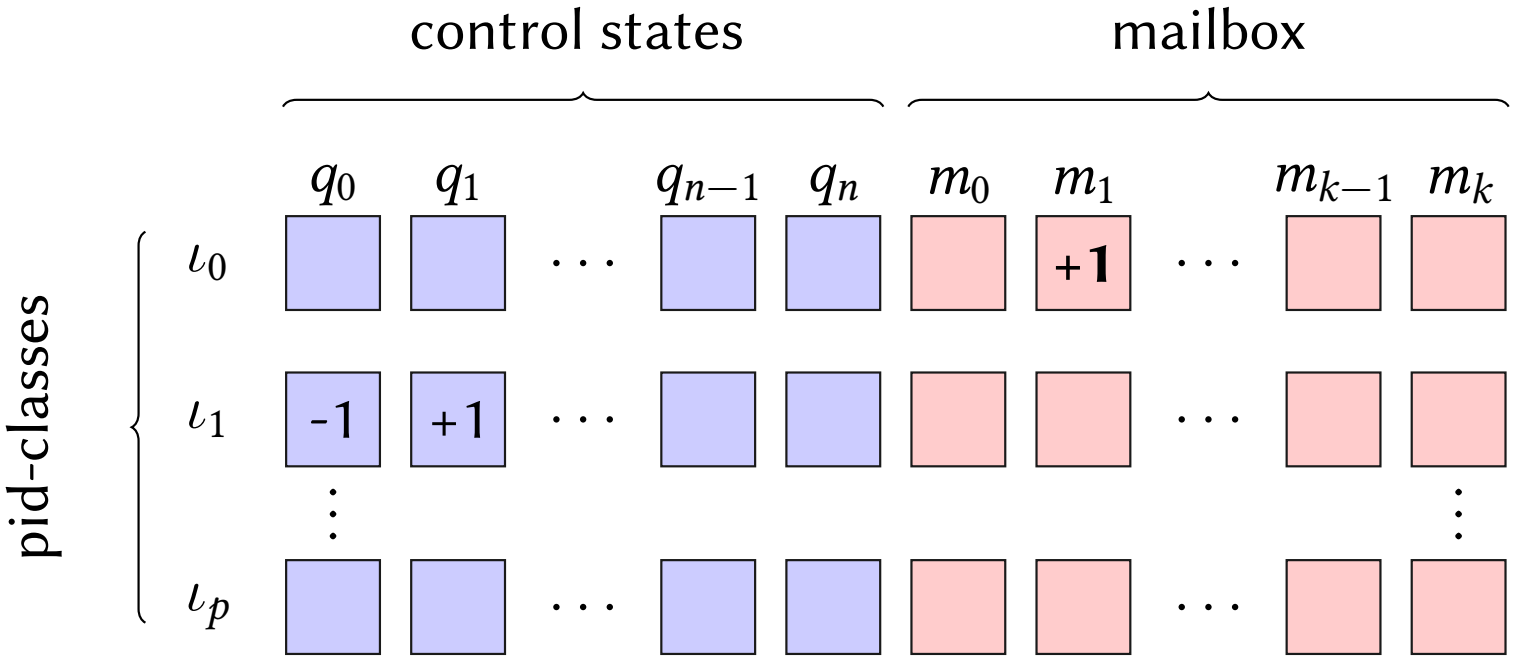


Underlying VAS



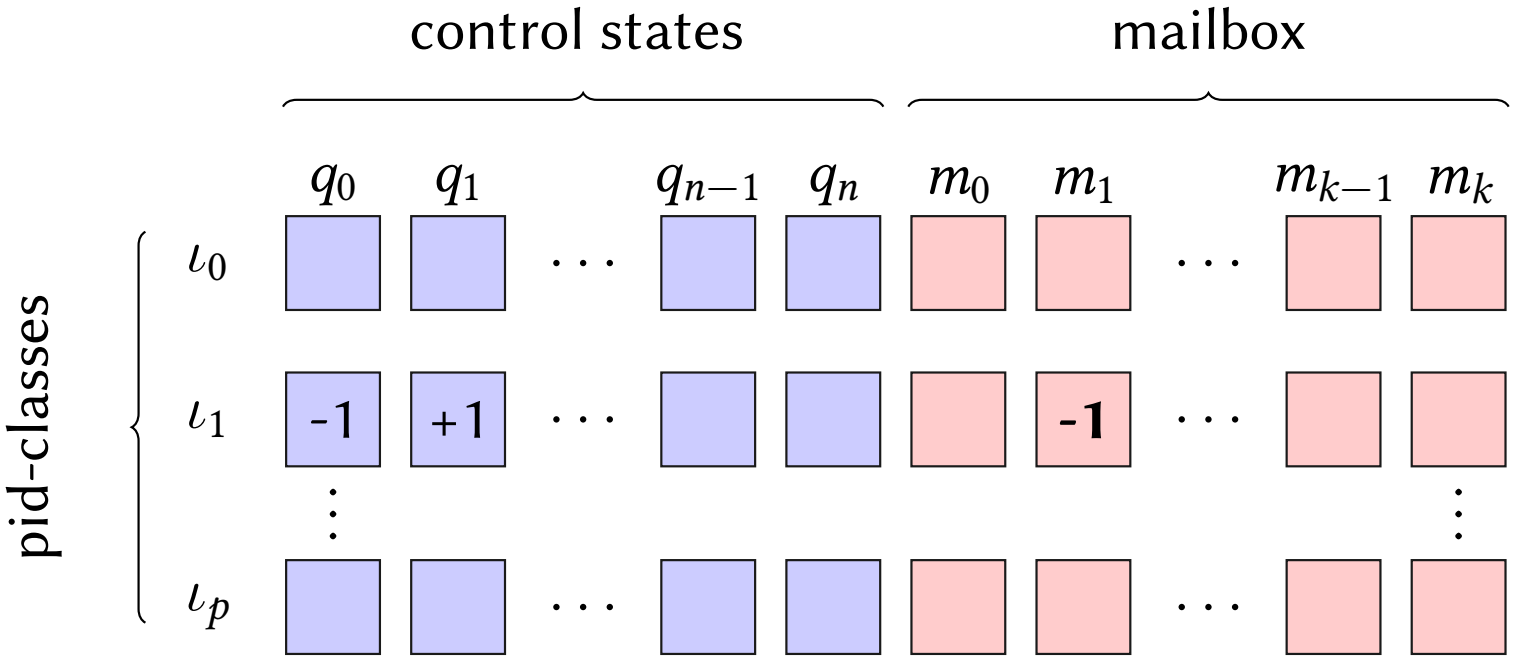
Seq. reduction $\iota_1 : q_0 \xrightarrow{\tau} q_1$

Underlying VAS



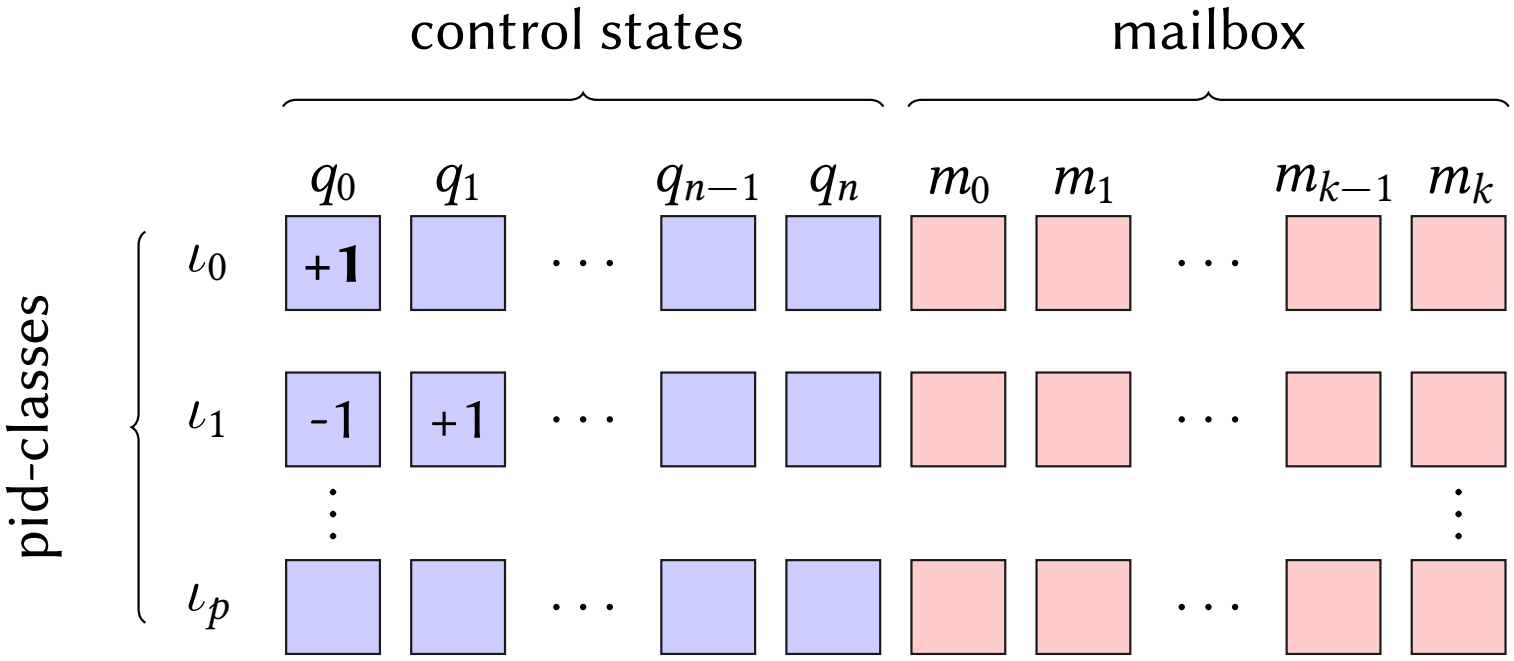
Send $\iota_1 : q_0 \xrightarrow{\iota_0 \,!\, m_1} q_1$

Underlying VAS



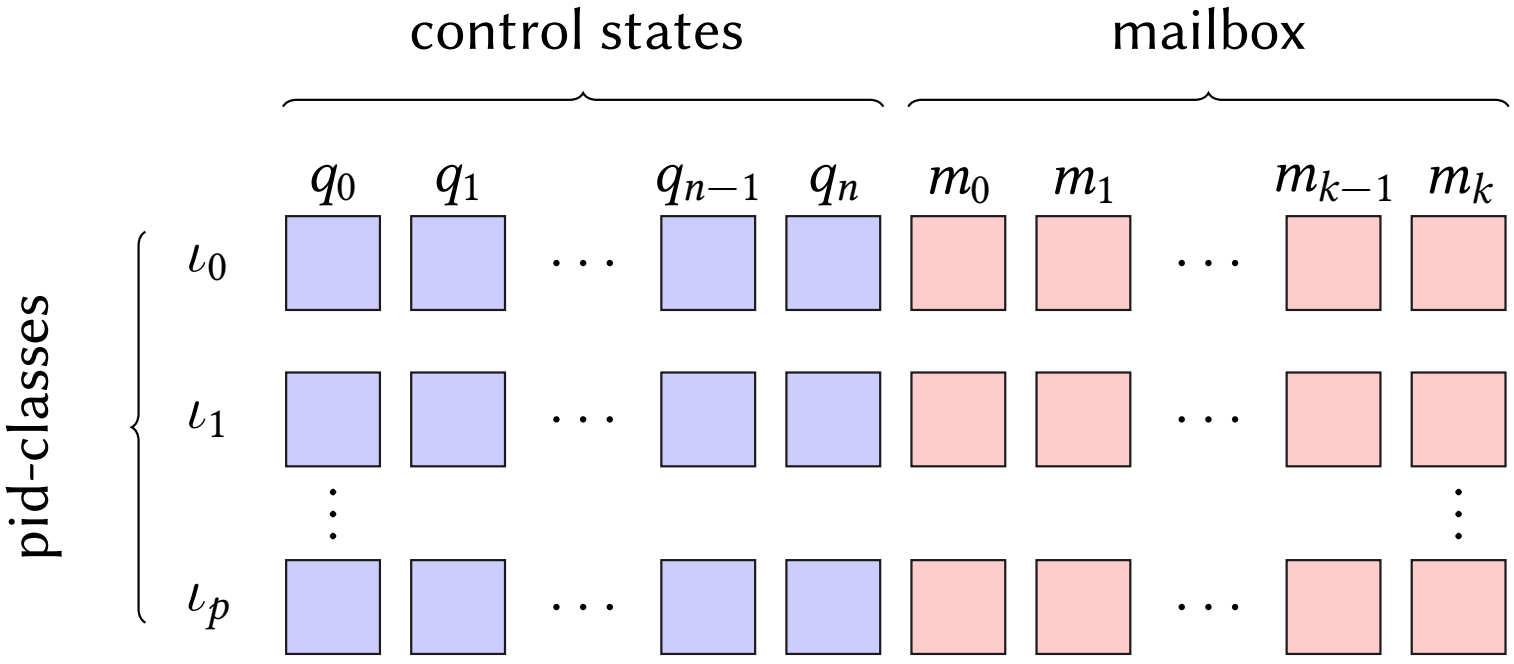
Receive $\iota_1 : q_0 \xrightarrow{?m_1} q_1$

Underlying VAS



Spawn $\quad \iota_1 : q_0 \xrightarrow{\nu \iota_0 . q_0} q_1$

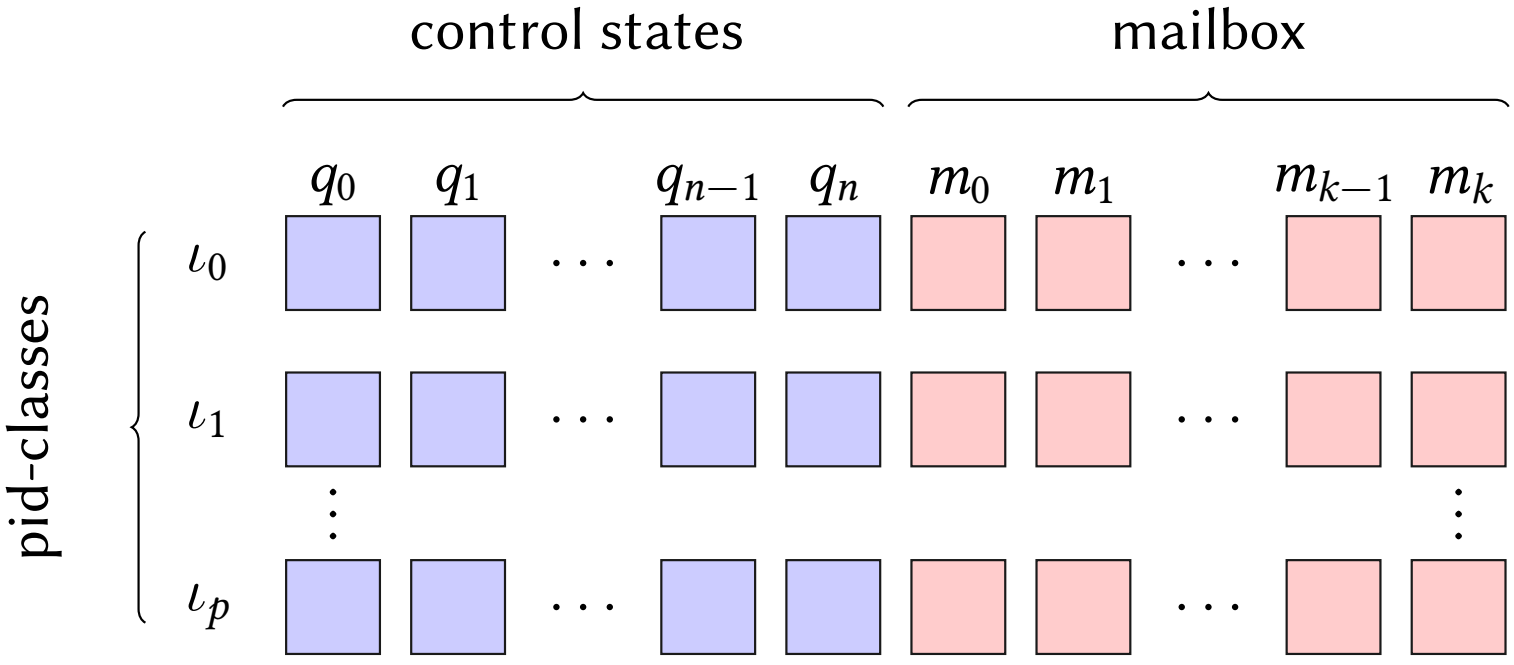
Underlying VAS



Interesting properties:

Mutual Exclusion $\mathbf{v}(\iota_1, q_1) < 2$

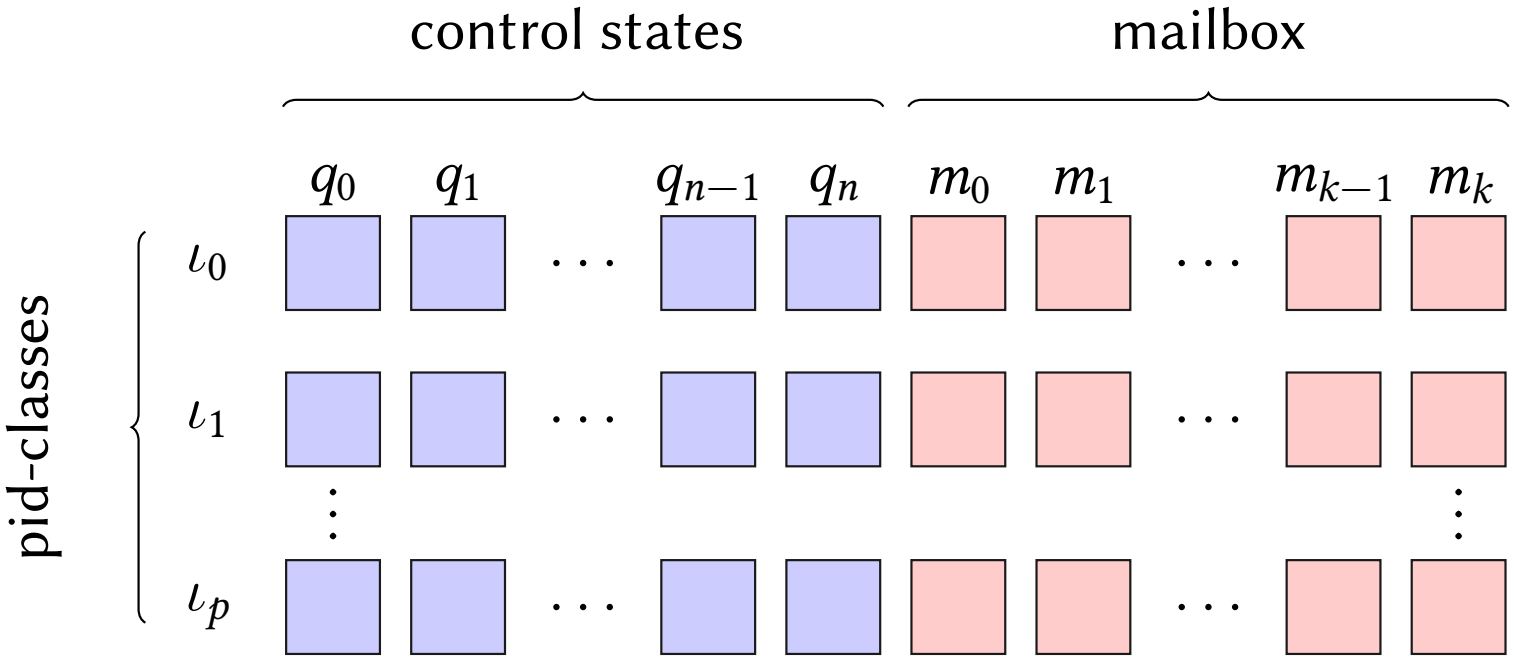
Underlying VAS



Interesting properties:

Absence of error $\mathbf{v}(\iota_1, q_1) = 0$

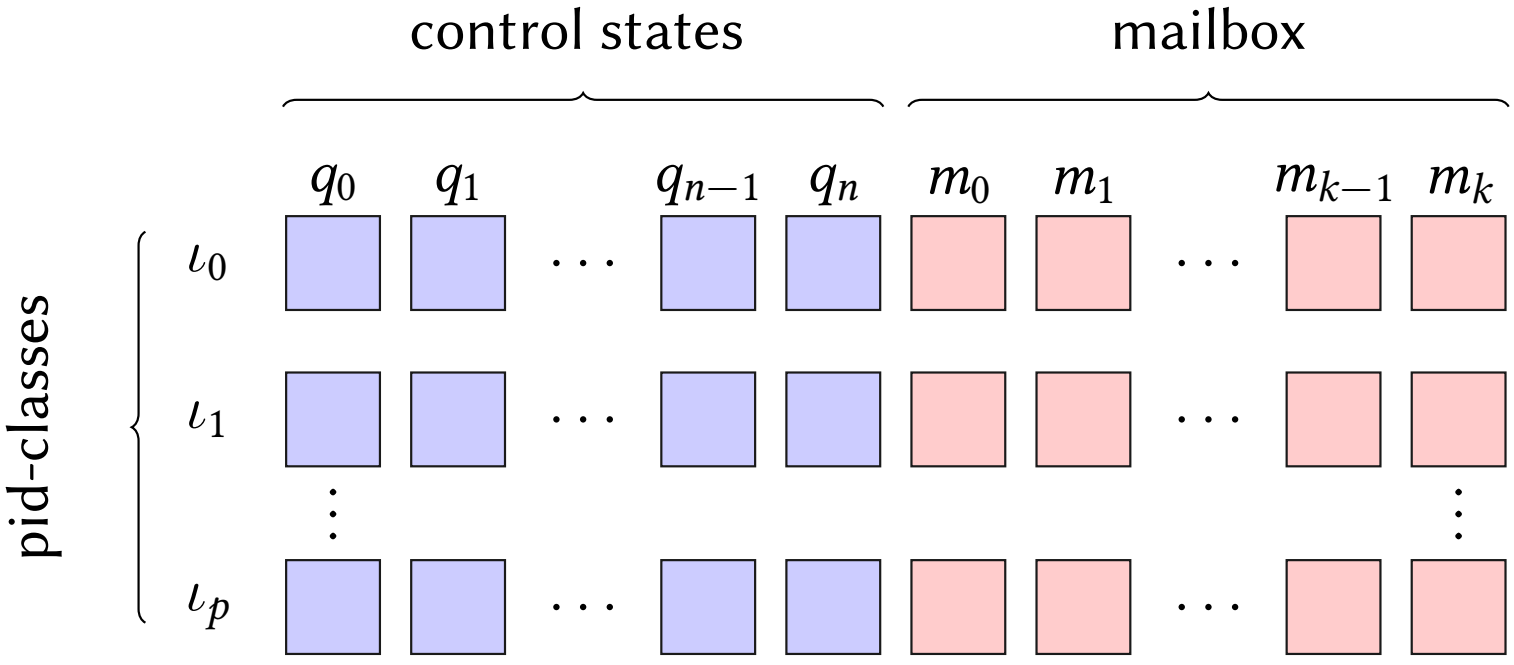
Underlying VAS



Interesting properties:

Bound on mailbox $\sum_{0 \leq i \leq k} \mathbf{v}(\iota_1, m_i) \leq B$

Underlying VAS



Interesting properties:

Coverability queries (EXPSPACE-complete)

Verification pathway

Our method:

- (Core) Erlang code as source
- A k-CFA-like analysis abstracts the control-flow
- The analysis produces an ACS which soundly approximates the program
- Model-check the ACS with VAS coverability engine (BFC)

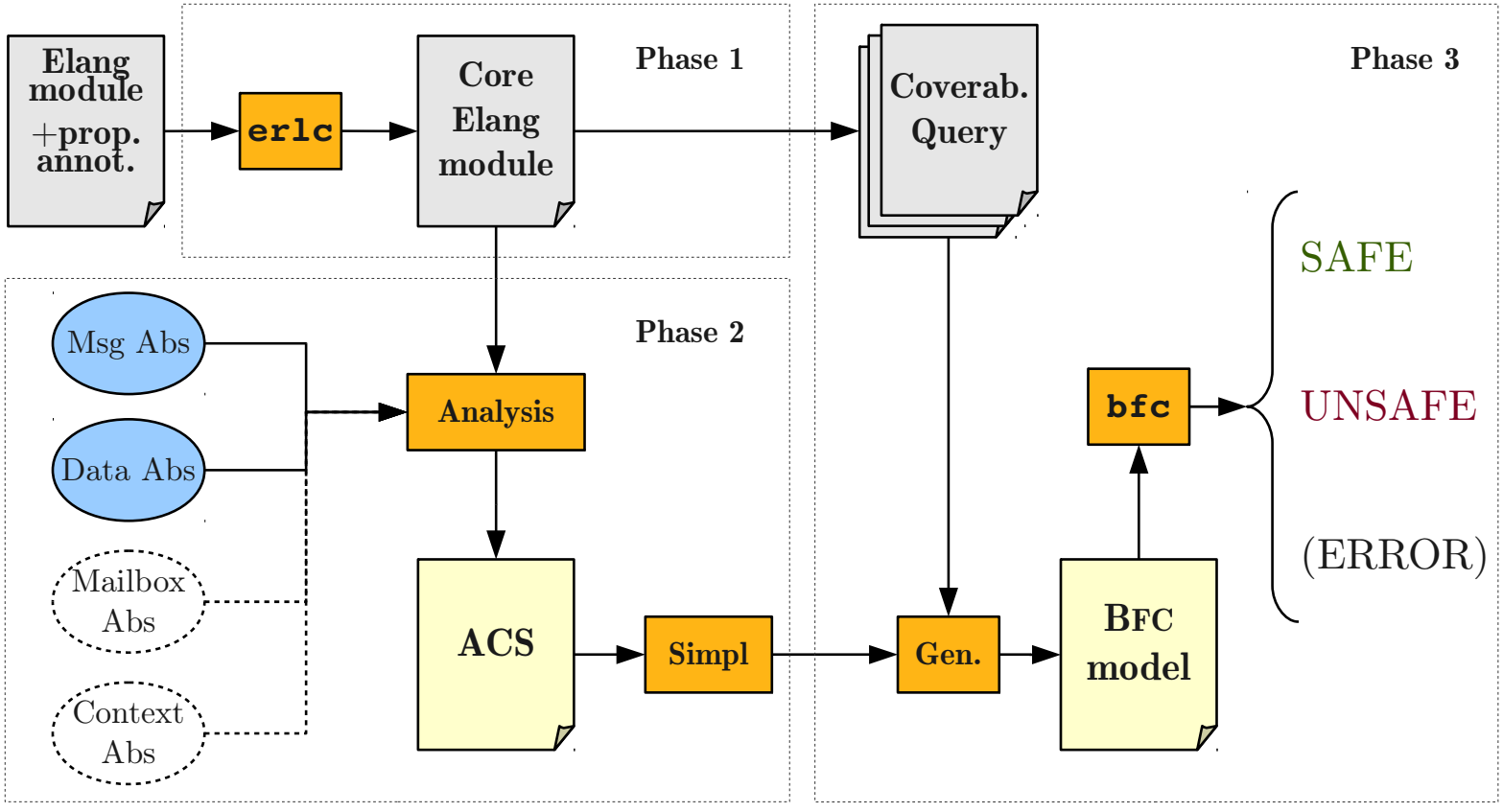
The analysis is parametric and can be tuned for accuracy.

In the paper

What's in the paper:

- A fully formal description of the parametric analysis
- Formal definition of ACS generation, with polynomial bounds on size of the model
- Formal proofs of soundness and termination
- A tool with some benchmarks (available as virtual machine and web-interface)

Soter



http://mjolnir.cs.ox.ac.uk/soter/

Future Work

Planned work includes

- support for full Erlang
- improve precision wrt:
  - process identities
  - stack behaviour
- find ways to handle open systems

Thanks!

http://mjolnir.cs.ox.ac.uk/soter/