goto; conference aarhus

Bloom: Using disorderly programming to build eventually-consistent distributed systems

Bill Marczak
UC Berkeley

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

gotocon.com

Why is dist. programming hard?

Why is dist. programming hard?

- Concurrency

Why is dist. programming hard?

- Concurrency
- Asynchrony
  - Delay
  - Re-ordering
  - Batching

Why is dist. programming hard?

- Concurrency
- Asynchrony
  - Delay
  - Re-ordering
  - Batching
- Partial failure

Why is dist. programming hard?

- Concurrency
- Asynchrony
  - Delay
  - Re-ordering
  - Batching
- Partial failure

Non-determinism

State of the art

- “Strong consistency” (e.g., 2PC, Paxos)
  - Global sync
  - High price
  - Prohibit interleavings; guaranteed consistency

State of the art

- “Strong consistency” (e.g., 2PC, Paxos)
  - Global sync
  - High price
  - Prohibit interleavings; guaranteed consistency
- “Loose” (eventual) consistency
  - Write program to avoid sync
  - Apply “ad-hoc genius”

State of the art

- “Strong consistency” (e.g., 2PC, Paxos)
  - Global sync
  - High price
  - Prohibit interleavings; guaranteed consistency
- “Loose” (eventual) consistency
  - Write program to avoid sync
  - Apply “ad-hoc genius”

Bloom

- New language + analysis to check for consistency (confluence)
- Internal DSL in Ruby

Bloom

- New language + analysis to check for consistency (confluence)
- Internal DSL in Ruby
- **Why is this in the DB track?**
  - Uses intuition from Datalog
    - recursive SQL
    - FO[LFP]
  - “Disorderly evaluation”: rule-based language; set abstraction

Outline

- **Flavor of BLOOM: Shopping cart**
- Static analysis
- A better shopping cart

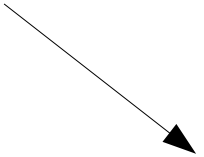
Bloom rules

```
collection          op          collection expr
```


`table`	Default persist
`scratch`	Default delete
`channel`	Remote scratch

Bloom rules

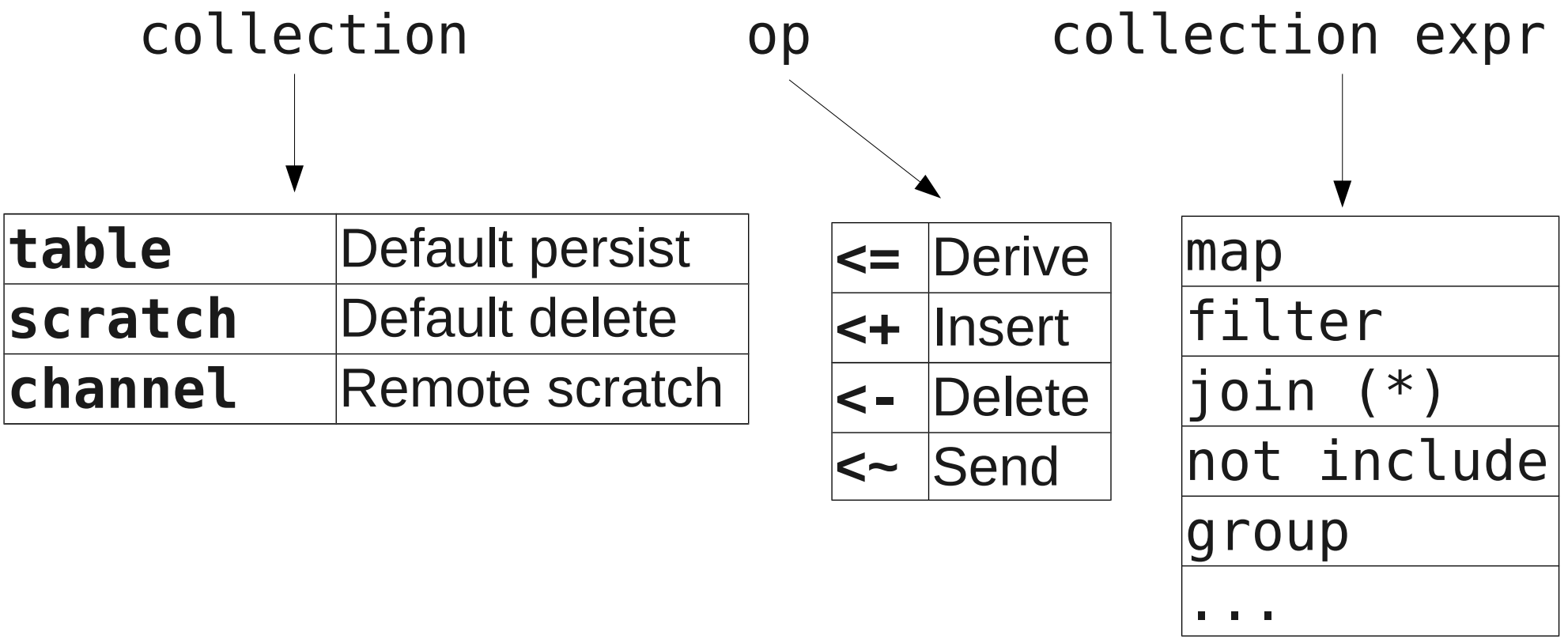
```
collection          op          collection expr
```

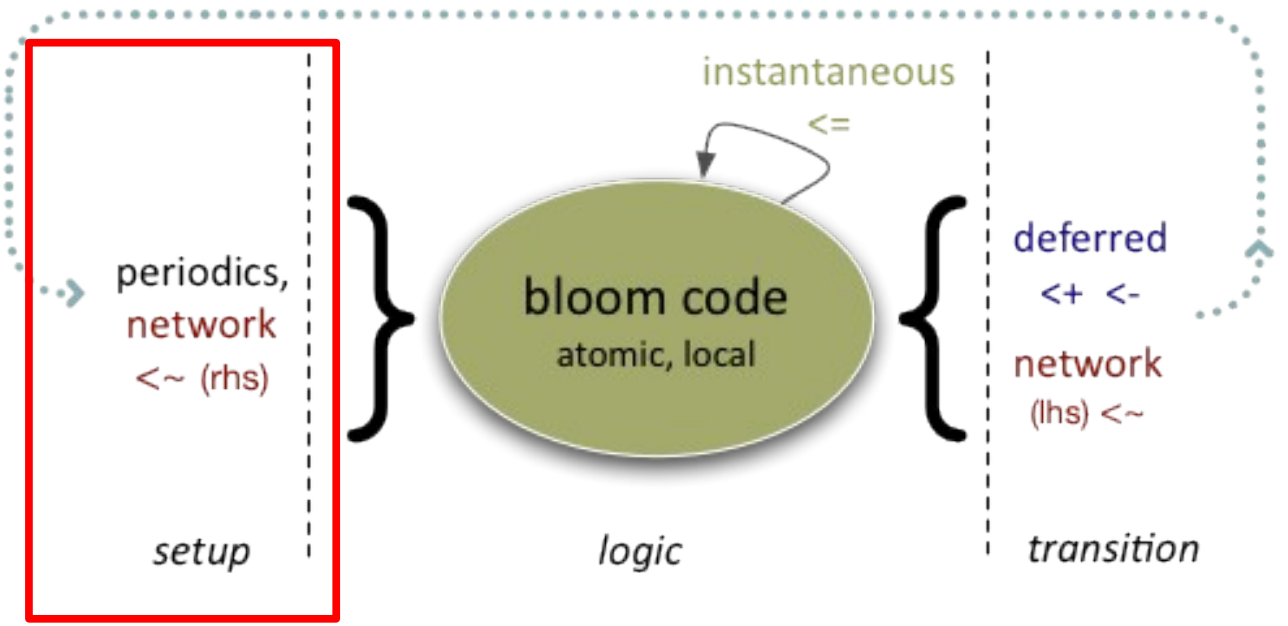
`table`	Default persist
`scratch`	Default delete
`channel`	Remote scratch

<=	Derive
<+	Insert
<-	Delete
<~	Send

Bloom rules

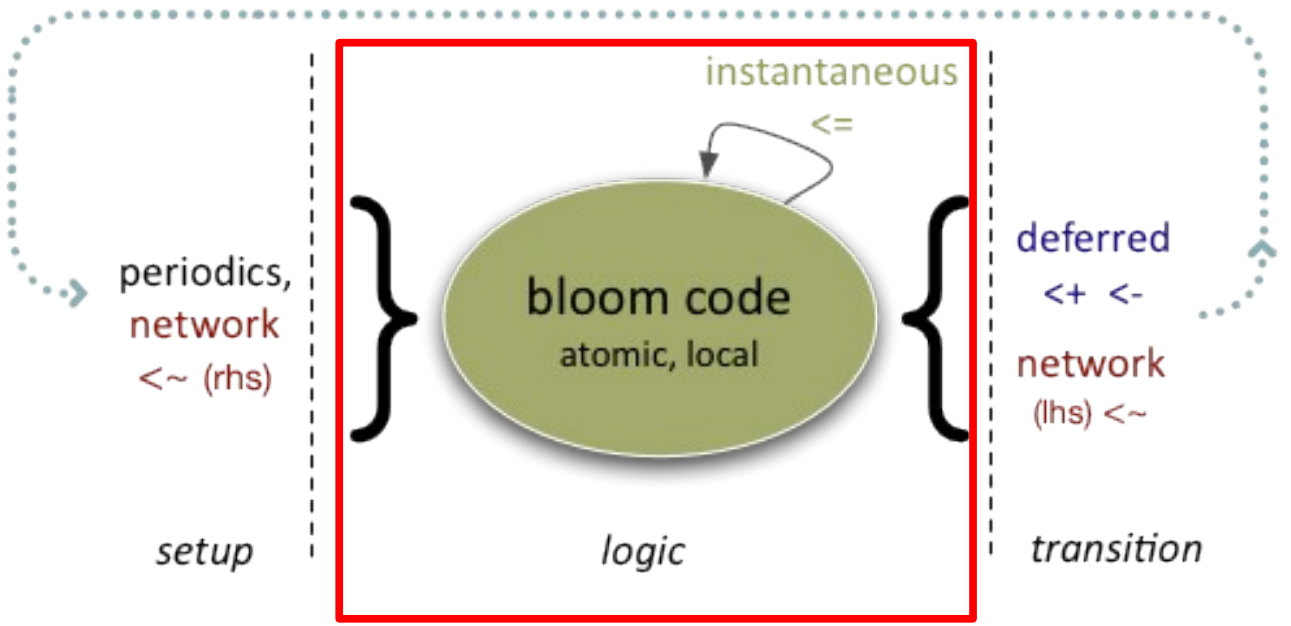

collection
op
collection expr
table
Default persist
scratch
Default delete
channel
Remote scratch
<=
Derive
<+
Insert
<-
Delete
<~
Send
map
filter
join (*)
not include
group
...

Operational Semantics



setup: scratches emptied, network messages placed in channels

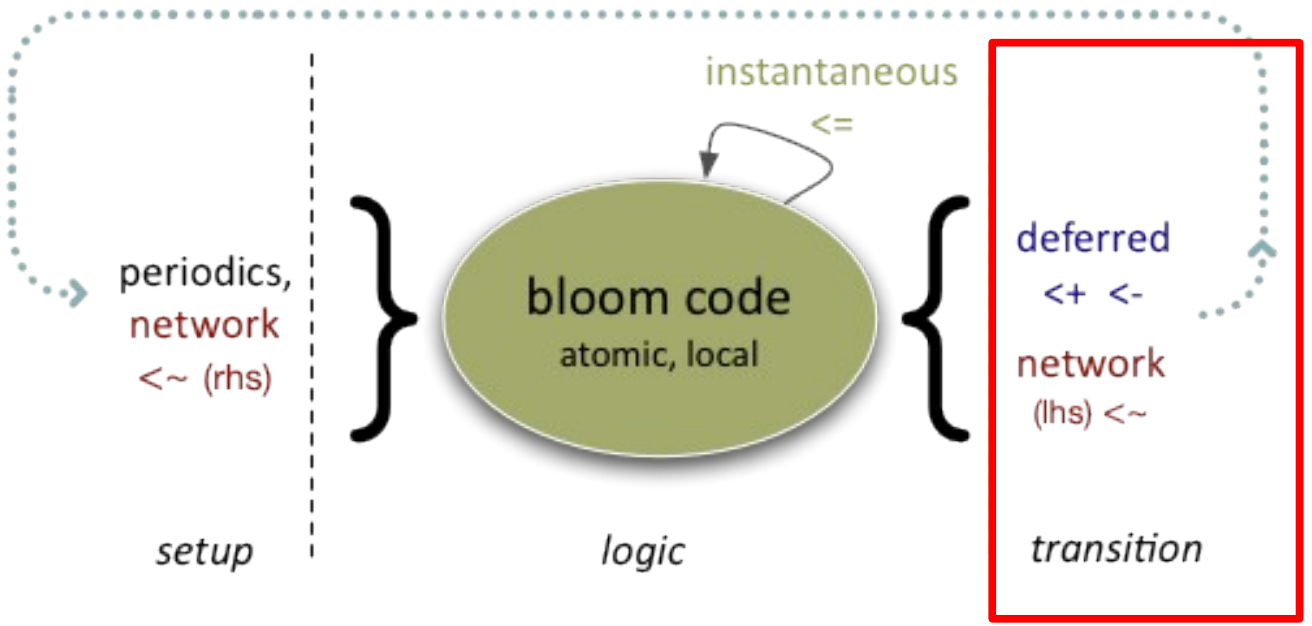
Operational Semantics



setup: scratches emptied, network messages placed in channels

logic: Bloom <= rules evaluated until fixpoint

Operational Semantics



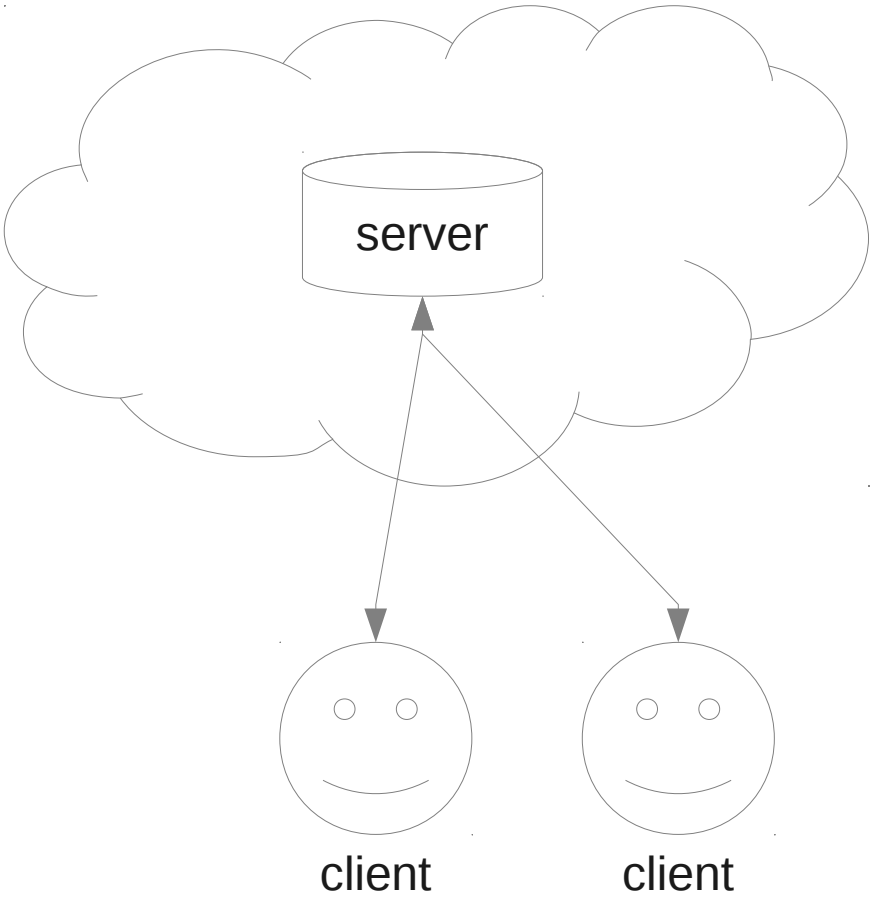
setup: scratches emptied, network messages placed in channels

logic: Bloom <= rules evaluated until fixpoint

transition: items derived by <+ and <- inserted/deleted from collections, items derived by <~ sent

Shopping cart

- Insert & delete items
- Check out
- Receive summary



Shopping cart

- **Abstract cart protocol**
  (messages exchanged between client & server)
- Concrete implementation of cart server
  - Abstract key/value store
  - Concrete key/value store

Abstract cart protocol

```
module CartProtocol
    state do









    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do




    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do




    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do

        channel :action_msg,   [:@server, :client, :reqid] =>
                               [:item, :action]









    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do

        channel :action_msg,    [:@server, :client, :reqid] =>
                                [:item, :action]









    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do

        channel :action_msg,    [:@server, :client, :reqid] =>
                                [:item, :action]









    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do

        channel :action_msg,   [:@server, :client, :reqid] =>
                               [:item, :action]
```

server: the IP address of the cart server
client: the IP address of the cart client
reqid: a unique identifier for the request
item: the shopping cart item to modify
action: add or delete from cart

```
    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do

        channel :action_msg,   [:@server, :client, :reqid] =>
                               [:item, :action]
```

at sign “@” indicates location of tuple

`server:` the IP address of the cart server
`client:` the IP address of the cart client
`reqid:` a unique identifier for the request
`item:` the shopping cart item to modify
`action:` add or delete from cart

right arrow indicates functional dependency

```
    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do

        channel :action_msg,   [:@server, :client, :reqid] =>
                               [:item, :action]



        channel :checkout_msg, [:@server, :client, :reqid]









    end
end
```

Abstract cart protocol

```
module CartProtocol
    state do

        channel :action_msg,   [:@server, :client, :reqid] =>
                               [:item, :action]



        channel :checkout_msg, [:@server, :client, :reqid]



        channel :response_msg, [:@client, :server, :item] =>
                               [:cnt]



    end
end
```

Shopping cart

- Abstract cart protocol
  (messages exchanged between client & server)
- Concrete implementation of cart server
  - **Abstract key/value store**
  - Concrete key/value store

Abstract KVS

```
module KVSProtocol
    state do









    end
end
```

Abstract KVS

```
module KVSProtocol
    state do

        interface input, :kvput, [:key] =>
                                 [:value]









    end
end
```

Abstract KVS

```
module KVSProtocol
    state do

        interface input, :kvput, [:key] =>
                                 [:value]

        interface input, :kvget, [:reqid] =>
                                 [:key]




    end
end
```

Abstract KVS

```
module KVSProtocol
    state do

        interface input, :kvput, [:key] =>
                                 [:value]

        interface input, :kvget, [:reqid] =>
                                 [:key]

        interface output, :kvget_response, [:reqid] =>
                                           [:key, :value]

    end
end
```

Shopping cart

- Abstract cart protocol
  (messages exchanged between client & server)
- **Concrete implementation of cart server**
  - Abstract key/value store
  - Concrete key/value store

Concrete Server

```
module DestructiveCart
     include CartProtocol
     include KVSProtocol




end
```

Concrete Server

```
module DestructiveCart
    include CartProtocol
    include KVSProtocol

    bloom :queueing do

    end

end
```

Concrete Server

```
module DestructiveCart
    include CartProtocol
    include KVSProtocol

    bloom :queueing do
        kvget <= action_msg {|a| [a.reqid, a.client] }

    end

end
```

Concrete Server

```
module DestructiveCart
    include CartProtocol
    include KVSProtocol

    bloom :queueing do
        kvget <= action_msg {|a| [a.reqid, a.client] }
        kvput <= action_msg do |a|
            if a.action == "Add" and not kvget_response.map{|b| b.key}.include? a.client
                [a.client, Array.new.push(a.item)]
            end
        end









    end









end
```

Concrete Server

```
module DestructiveCart
    include CartProtocol
    include KVSProtocol

    bloom :queueing do
        kvget <= action_msg {|a| [a.reqid, a.client] }
        kvput <= action_msg do |a|
            if a.action == "Add" and not kvget_response.map{|b| b.key}.include? a.client
                [a.client, Array.new.push(a.item)]
            end
        end


        temp :old_state <= (kvget_response * action_msg).pairs(:key => :client)
        kvput <= old_state do |b, a|
            if a.action == "Add"
                [a.client, (b.value.clone.push(a.item))]
            elsif a.action == "Del"
                [a.client, delete_one(b.value, a.item)]
            end
        end
    end




end
```

Concrete Server

```
module DestructiveCart
    include CartProtocol
    include KVSProtocol

    bloom :queueing do
        kvget <= action_msg {|a| [a.reqid, a.client] }
        kvput <= action_msg do |a|
            if a.action == "Add" and not kvget_response.map{|b| b.key}.include? a.client
                [a.client, Array.new.push(a.item)]
            end
        end


        temp :old_state <= (kvget_response * action_msg).pairs(:key => :client)
        kvput <= old_state do |b, a|
            if a.action == "Add"
                [a.client, (b.value.clone.push(a.item))]
            elsif a.action == "Del"
                [a.client, delete_one(b.value, a.item)]
            end
        end
    end

    bloom :finish do
        kvget <= checkout_msg{|c| [c.reqid, c.client] }
        temp :lookup <= (kvget_response * checkout_msg).pairs(:key => :client)
    end
end
```

Shopping cart

- Abstract cart protocol
  (messages exchanged between client & server)
- Concrete implementation of cart server
  - Abstract key/value store
  - **Concrete key/value store**

Concrete KVS

```
module BasicKVS
    include KVSProtocol

end
```

Concrete KVS

```
module BasicKVS
    include KVSProtocol

    state do
        table :kvstate, [:key] => [:value]
    end

end
```

Concrete KVS

```
module BasicKVS
    include KVSProtocol

    state do
        table :kvstate, [:key] => [:value]
    end

    bloom :mutate do
        kvstate <+ kvput {|s| [s.key, s.value]}
        kvstate <- (kvstate * kvput).lefts(:key => :key)
    end




end
```

Concrete KVS

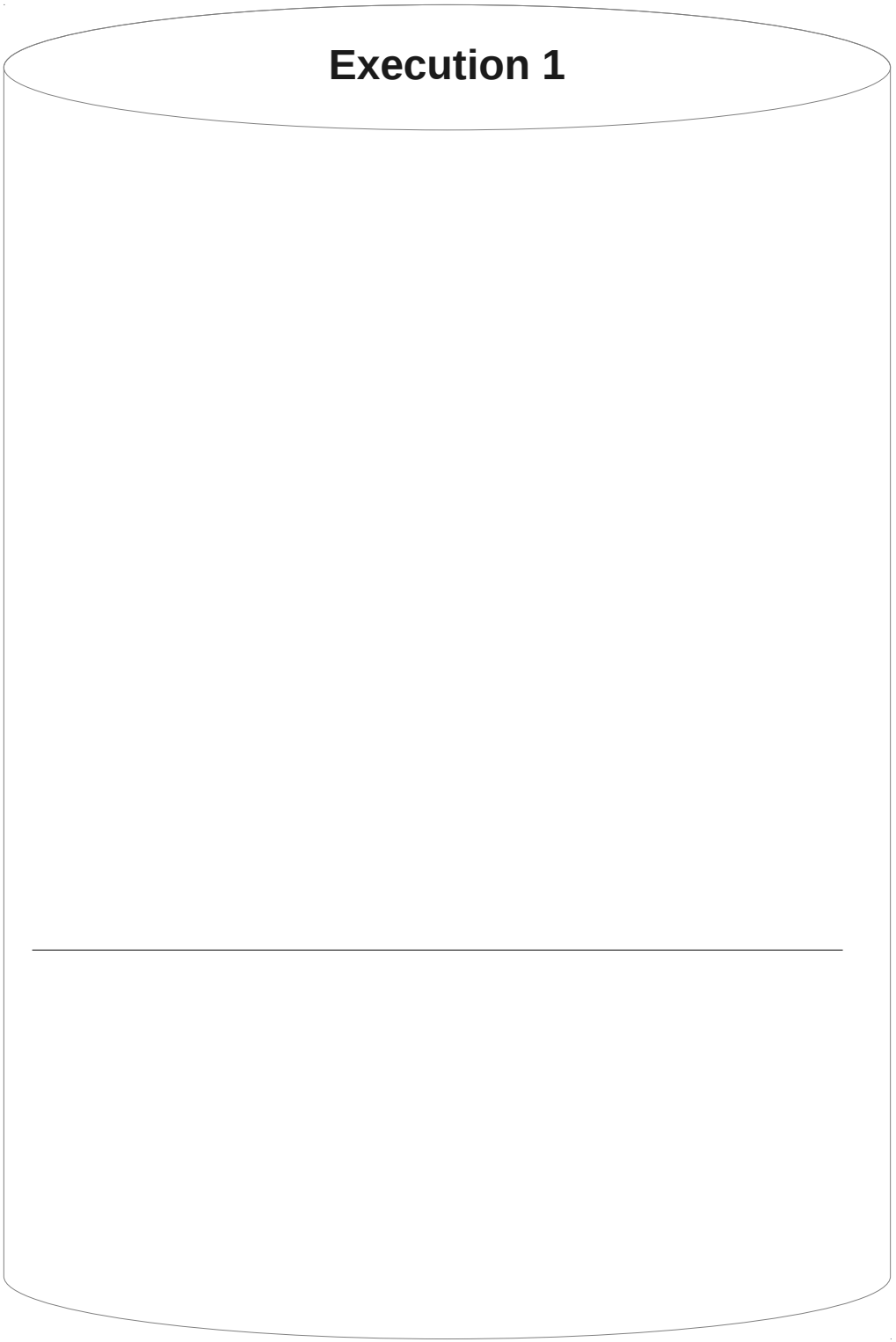
```
module BasicKVS
    include KVSProtocol

    state do
        table :kvstate, [:key] => [:value]
    end

    bloom :mutate do
        kvstate <+ kvput {|s| [s.key, s.value]}
        kvstate <- (kvstate * kvput).lefts(:key => :key)
    end

    bloom :get do
        temp :getj <= (kvget * kvstate).pairs(:key => :key)

        kvget_response <= getj { |g, t| [g.reqid, t.key, t.value] }
    end

end
```

Ad-hoc genius analysis

Ad-hoc genius analysis

action_msg			
client	**reqid**	**item**	**action**
“Alice”	1	“2TB HD”	“add”
“Alice”	2	“2TB HD”	“del”
“Alice”	3	“128GB SSD”	“add”

checkout_msg	
client	**reqid**
“Alice”	4


Execution 1

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**3**	“128GB SSD”	“add”

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**3**	“128GB SSD”	“add”

---

response_msg		
client	**item**	**cnt**
“Alice”	“2TB HD”	1

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**3**	“128GB SSD”	“add”

---

response_msg		
client	**item**	**cnt**
“Alice”	“2TB HD”	1

Execution 2

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**3**	“128GB SSD”	“add”

---

response_msg		
client	**item**	**cnt**
“Alice”	“2TB HD”	1

Execution 2

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**1**	“2TB HD”	“add”
“Alice”	**3**	“128GB SSD”	“add”

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**3**	“128GB SSD”	“add”

---

response_msg		
client	**item**	**cnt**
“Alice”	“2TB HD”	1

Execution 2

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**1**	“2TB HD”	“add”
“Alice”	**3**	“128GB SSD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

Execution 1

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**1**	“2TB HD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**3**	“128GB SSD”	“add”

---

response_msg		
client	**item**	**cnt**
“Alice”	“2TB HD”	1

Execution 2

action_msg			
client	**reqid**	**item**	**action**
“Alice”	**2**	“2TB HD”	“del”
“Alice”	**1**	“2TB HD”	“add”
“Alice”	**3**	“128GB SSD”	“add”

checkout_msg	
client	**reqid**
“Alice”	**4**

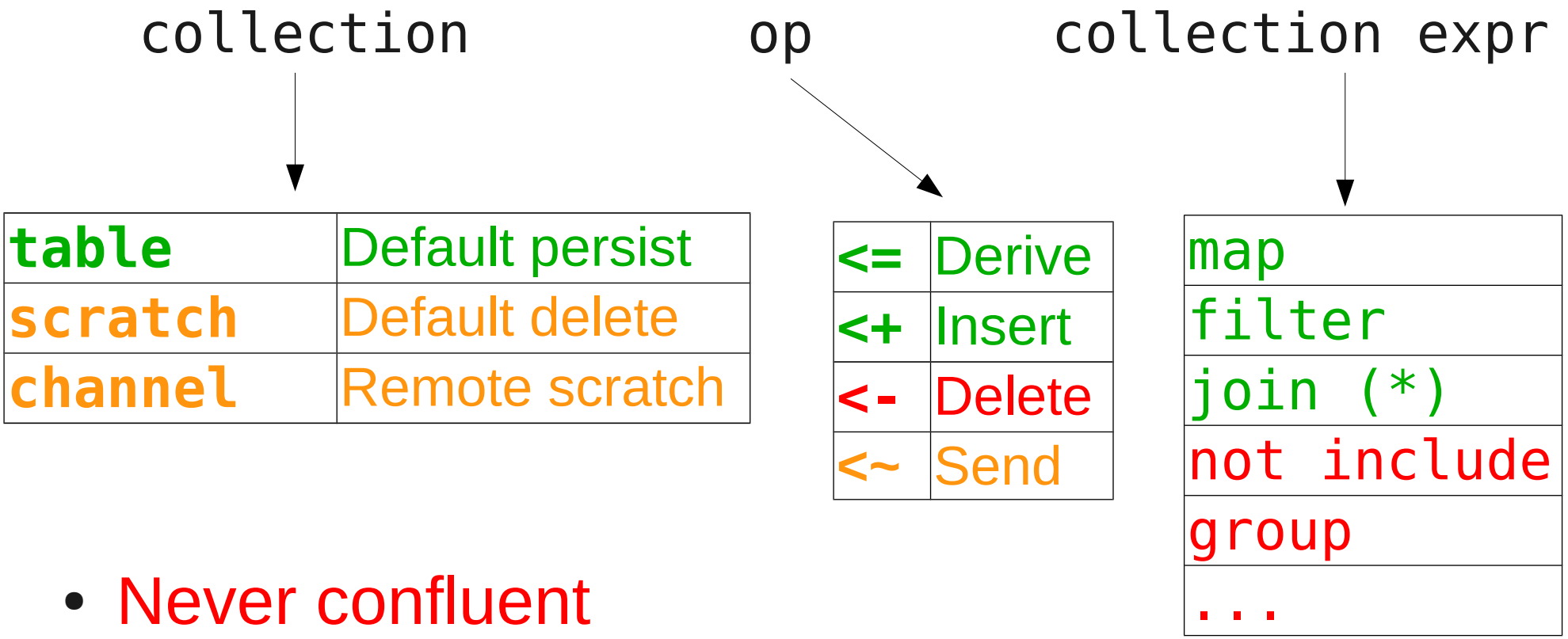
---

response_msg		
client	**item**	**cnt**
“Alice”	“2TB HD”	1
“Alice”	“128GB SSD”	1

Ad-hoc genius analysis

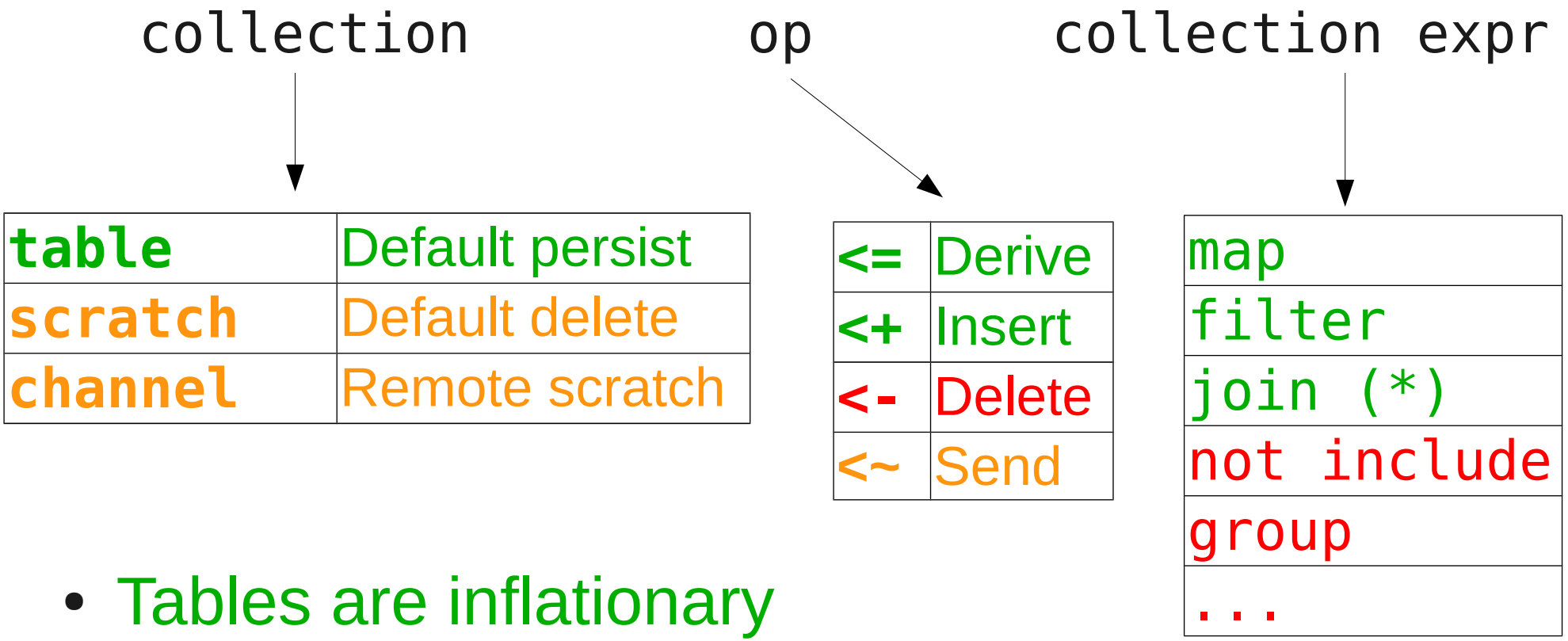
- What went wrong?
  - Delete before add?
  - `checkout_msg` in the middle of `action_msg`?

Static analysis



- Never confluent
- Confluent if guarded by table
- Always confluent

Static analysis



- Tables are inflationary
- map, filter, join are homomorphisms
- <=, <+ are commutative, associative

Abstract cart protocol, revisited

```
module CartProtocol
    state do

        channel :action_msg,   [:@server, :client, :reqid] =>
                               [:item, :action]


        channel :checkout_msg, [:@server, :client, :reqid]


        channel :response_msg, [:@client, :server, :item] =>
                               [:cnt]


    end
end
```

Abstract KVS, revisited

```
module KVSProtocol
    state do

        interface input, :kvput, [:key] =>
                                 [:value]

        interface input, :kvget, [:reqid] =>
                                 [:key]

        interface output, :kvget_response, [:reqid] =>
                                           [:key, :value]

    end
end
```

KVS, revisited

```
module BasicKVS
    include KVSProtocol

    state do
        table :kvstate, [:key] => [:value]
    end

    bloom :mutate do
        kvstate <+ kvput {|s| [s.key, s.value]}
        kvstate <- (kvstate * kvput).lefts(:key => :key)
    end

    bloom :get do
        temp :getj <= (kvget * kvstate).pairs(:key => :key)

        kvget_response <= getj do |g, t|
            [g.reqid, t.key, t.value]
        end
    end

end
```

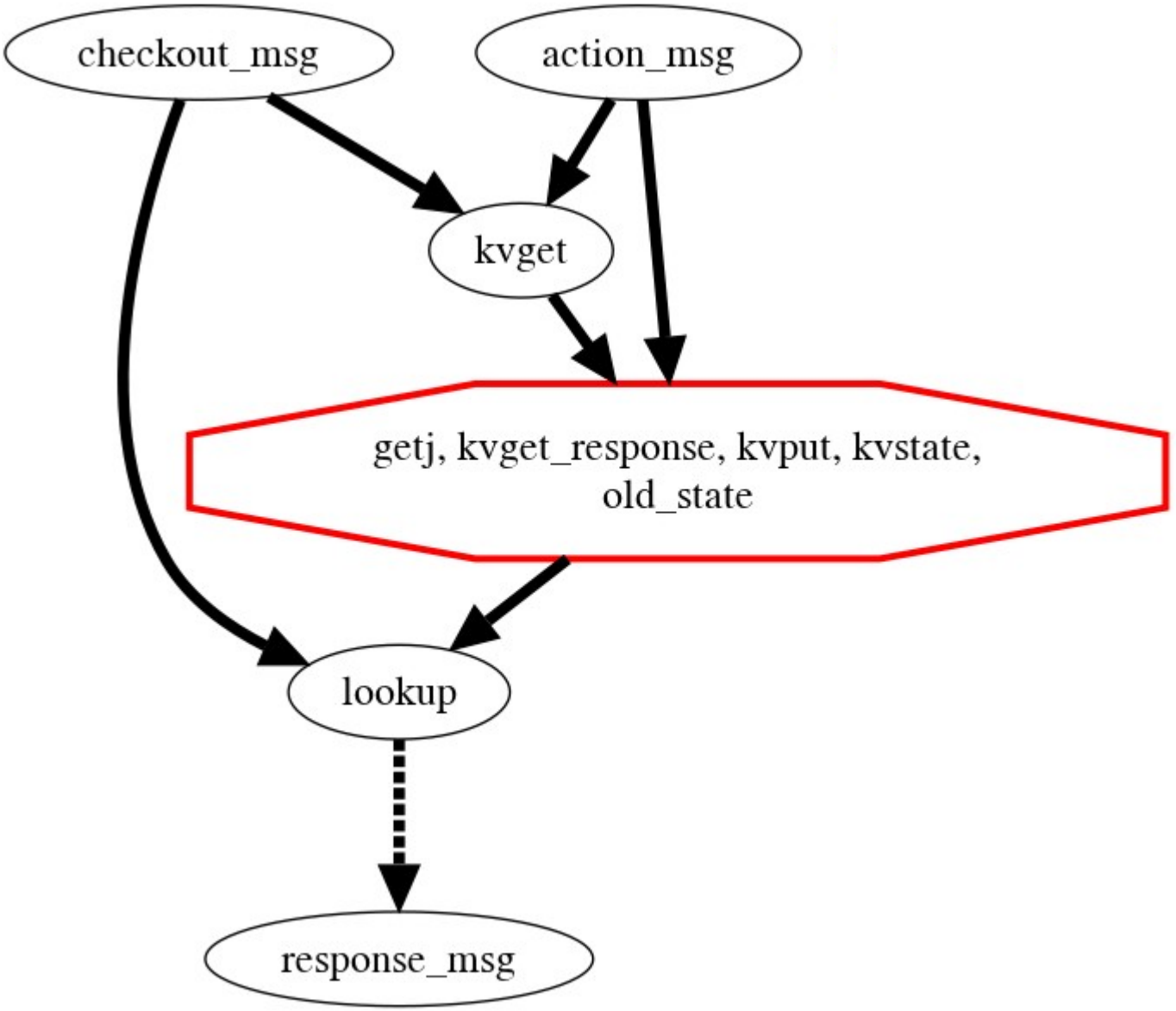
Cart, revisited

[…]

```
kvput <= action_msg do |a|
  if a.action=="Add" and not kvget_response{|b| b.key}.include? a.client
    [a.client, Array.new.push(a.item)]
  end
end
```

[…]

Static analysis



Solutions?

- Total ordering on request ID?
  - What if clients can modify cart from multiple locations?
  - Inconsistent w/ “disorderly” philosophy

Solutions?

- Total ordering on request ID?
  - What if clients can modify cart from multiple locations?
  - Inconsistent w/ “disorderly” philosophy
- “Disorderly” cart
  - Don't use ordering
  - Shopping cart kind of like a set (except it's a multiset)

Disorderly cart

```
module DisorderlyCart
  include CartProtocol









end
```

Disorderly cart

```
module DisorderlyCart
  include CartProtocol

  state do
    table :cart_action, [:client, :reqid] => [:item, :action]
    scratch :action_cnt, [:client, :item, :action] => [:cnt]
  end









end
```

Disorderly cart

```
module DisorderlyCart
  include CartProtocol

  state do
    table :cart_action, [:client, :reqid] => [:item, :action]
    scratch :action_cnt, [:client, :item, :action] => [:cnt]
  end

  bloom :saved do









  end
end
```

Disorderly cart

```
module DisorderlyCart
  include CartProtocol

  state do
    table :cart_action, [:client, :reqid] => [:item, :action]
    scratch :action_cnt, [:client, :item, :action] => [:cnt]
  end

  bloom :saved do
    cart_action <= action_msg {|c| [c.client, c.reqid, c.item, c.action]}









  end
end
```

Disorderly cart

```
module DisorderlyCart
  include CartProtocol

  state do
    table :cart_action, [:client, :reqid] => [:item, :action]
    scratch :action_cnt, [:client, :item, :action] => [:cnt]
  end

  bloom :saved do
    cart_action <= action_msg {|c| [c.client, c.reqid, c.item, c.action]}

    temp :checkout_acts <= (checkout_msg * cart_action).rights

    action_cnt <= checkout_acts.group([cart_action.client, cart_action.item,
                                       cart_action.action], count(cart_action.reqid))



  end
end
```

Disorderly cart

```
module DisorderlyCart
  include CartProtocol

  state do
    table :cart_action, [:client, :reqid] => [:item, :action]
    scratch :action_cnt, [:client, :item, :action] => [:cnt]
  end

  bloom :saved do
    cart_action <= action_msg {|c| [c.client, c.reqid, c.item, c.action]}

    temp :checkout_acts <= (checkout_msg * cart_action).rights

    action_cnt <= checkout_acts.group([cart_action.client, cart_action.item,
                                       cart_action.action], count(cart_action.reqid))

    response_msg <~ (checkout_msg * action_cnt).rights(:client => :client)

  end
end
```

Static Analysis

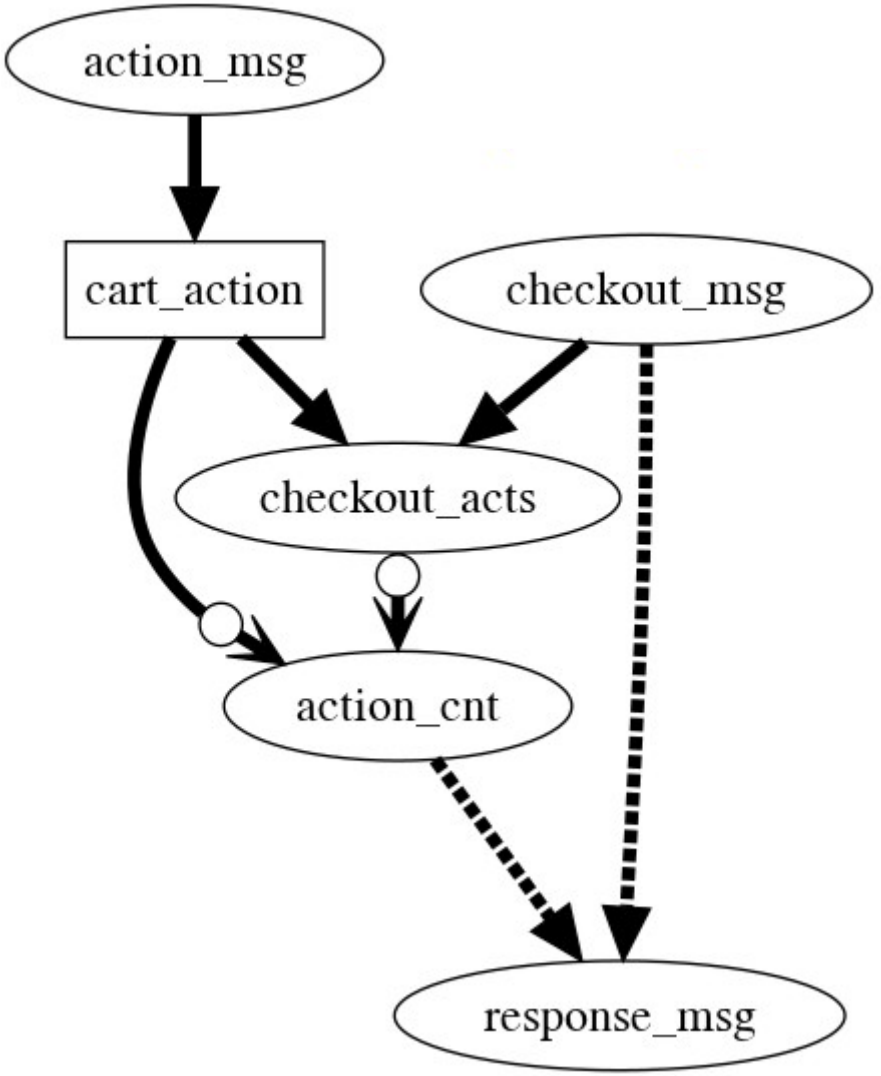
Cart, revisited

```
module DisorderlyCart
  include CartProtocol

  state do
    table :cart_action, [:client, :reqid] => [:item, :action]
    scratch :action_cnt, [:client, :item, :action] => [:cnt]
  end

  bloom :saved do
    cart_action <= action_msg {|c| [c.client, c.reqid, c.item, c.action]}

    temp :checkout_acts <= (checkout_msg * cart_action).rights

    action_cnt <= checkout_acts.group([cart_action.client, cart_action.item,
                                cart_action.action], count(cart_action.reqid))

    response_msg <~ (checkout_msg * action_cnt).rights(:client => :client)

  end
end
```

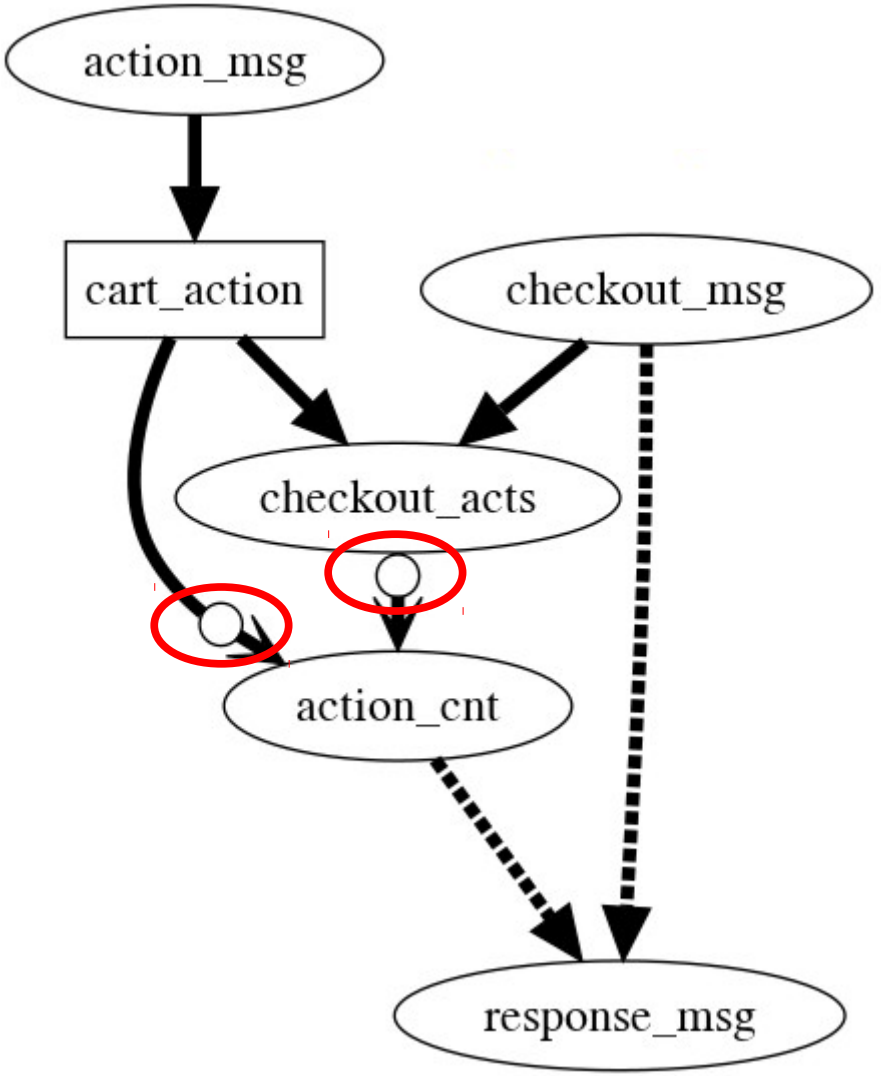
Abstract cart protocol, revisited

```
module CartProtocol
    state do

        channel :action_msg,    [:@server, :client, :reqid] =>
                                [:item, :action]



        channel :checkout_msg, [:@server, :client, :reqid]



        channel :response_msg, [:@client, :server, :item] =>
                                [:cnt]



    end
end
```

Static Analysis



Static Analysis



Current Work

- Adding more commutative data structures and functions
- Minimizing execution traces for debugging
- Fault tolerance support
- Optimization of Bloom programs

Other cool stuff

- REBL: Bloom REPL
- Deployment
  - Local thread/process
  - EC2
- TokyoCabinet, Zookeeper collections
- Visualizer for debugging
- BUST: REST client & server interfaces
- **Lots** of examples and documentation!

Check it out!

Bud: “Bloom under development”

```
gem install bud
```

```
http://github.org/bloom-lang
```

goto; conference aarhus

Bloom: Using disorderly programming to build eventually-consistent distributed systems

Bill Marczak
UC Berkeley

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

gotocon.com