Declarative programming over eventually consistent data stores

CMPS290S, Fall 2018
Lindsey Kuper
November 9, 2018

main idea

Hey, what if we took Burckhardt and co.'s RDT specification framework[1]... ...and *turned it into a programming language*?!

KC Sivaramakrishnan

[1] you know, the *vis* relation and all that stuff, as seen in Gotsman et al.'s POPL '14 paper, Burckhardt's book, the Viotti and Vukolić survey, and probably a bunch of other places

main idea

In particular, what if we turned it into a *contract language* for specifying the *consistency semantics* of data stores and operations on them?

KC Sivaramakrishnan

main idea

In particular, what if we turned it into a *contract language* for specifying the *consistency semantics* of data stores and operations on them?

…and implemented it in Haskell?

KC Sivaramakrishnan

main idea

In particular, what if we turned it into a *contract language* for specifying the *consistency semantics* of data stores and operations on them?

…and implemented it in Haskell?

…and stuck it on top of Cassandra?

KC Sivaramakrishnan

main idea

KC Sivaramakrishnan

In particular, what if we turned it into a *contract language* for specifying the *consistency semantics* of data stores and operations on them?

…and implemented it in Haskell?

…and stuck it on top of Cassandra?

yeah sounds good let’s do that

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sel1 $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

operations

every op gets a *history* of known updates to the object as its first argument

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

summarization: every now and then (how often?) the history gets boiled down

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

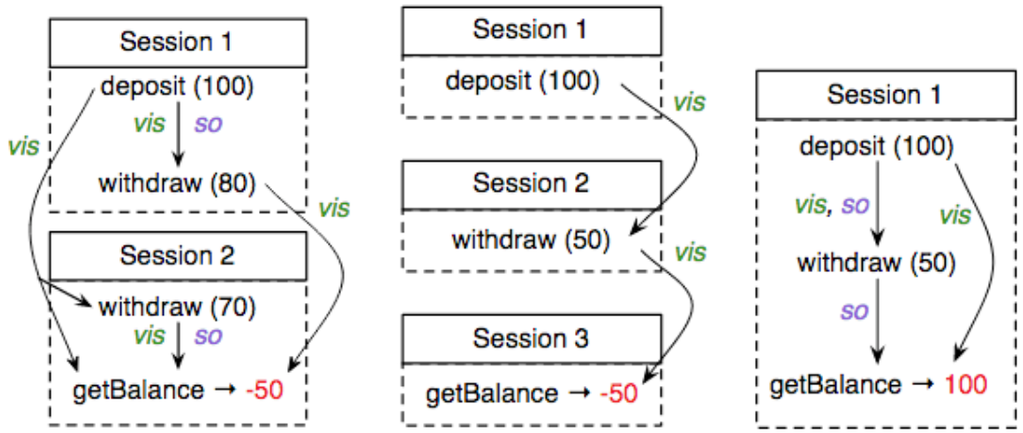
summarization: every now and then (how often?) the history gets boiled down

Q for class: how is summarization done for this particular RDT?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

let's play "name that anomaly"



operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

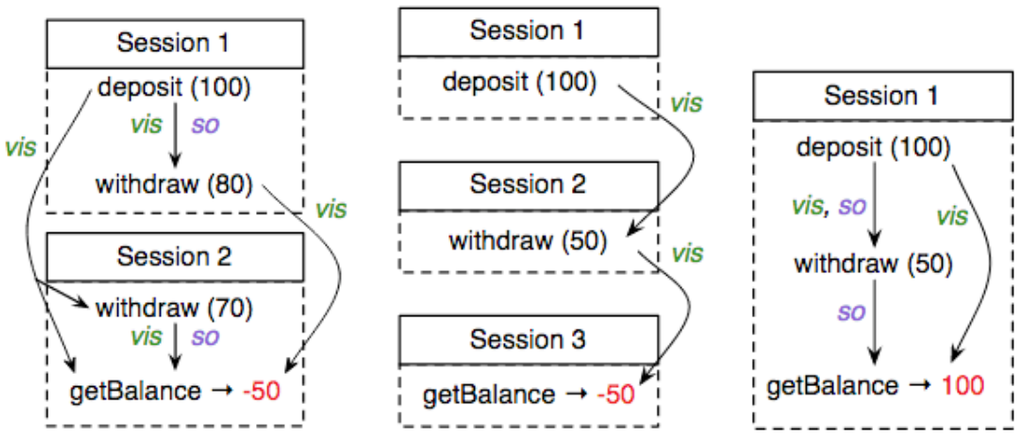
summarization: every now and then (how often?) the history gets boiled down

Q for class: how is summarization done for this particular RDT?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

let's play "name that anomaly"



violation of *integrity invariant* (balance ≥ 0)
fixable w/ strong consistency
(or token system, as in the CISE paper)

operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

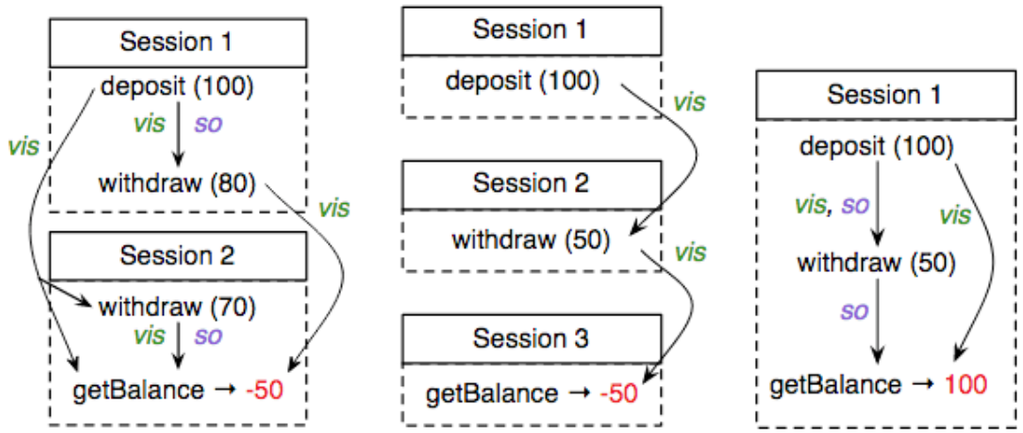
summarization: every now and then (how often?) the history gets boiled down

Q for class: how is summarization done for this particular RDT?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

let's play "name that anomaly"



operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

summarization: every now and then (how often?) the history gets boiled down

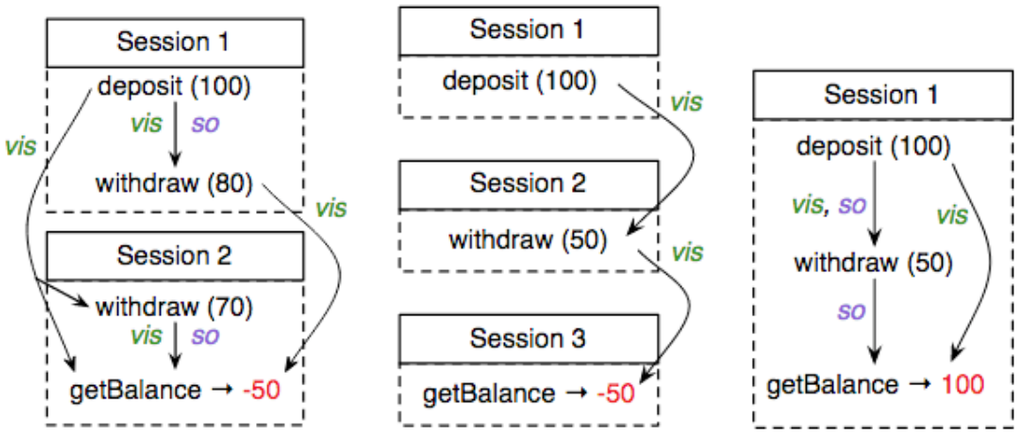
Q for class: how is summarization done for this particular RDT?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

operations

let's play "name that anomaly"



violation of *causality*
fixable w/ causal consistency

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

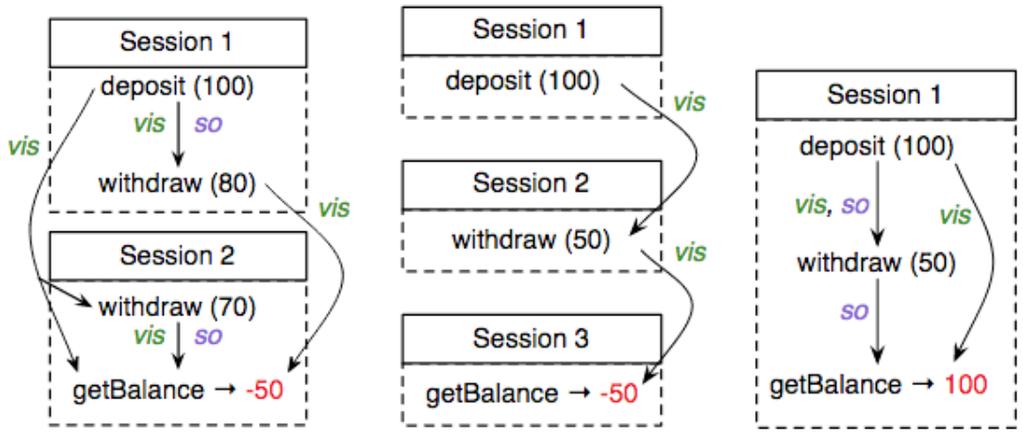
summarization: every now and then (how often?) the history gets boiled down

Q for class: how is summarization done for this particular RDT?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

let's play "name that anomaly"



operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

summarization: every now and then (how often?) the history gets boiled down

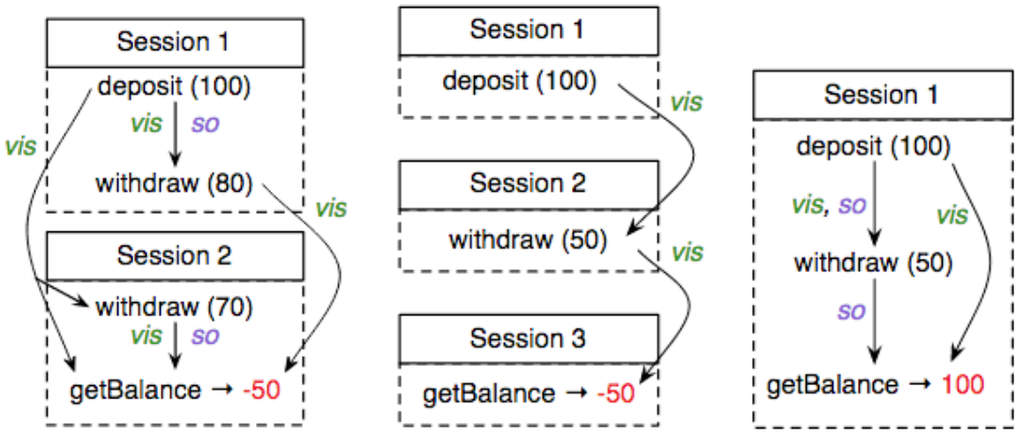
Q for class: how is summarization done for this particular RDT?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

operations

let's play "name that anomaly"



violation of *RYW session guarantee*
(and also causality)
fixable w/ causal consistency

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

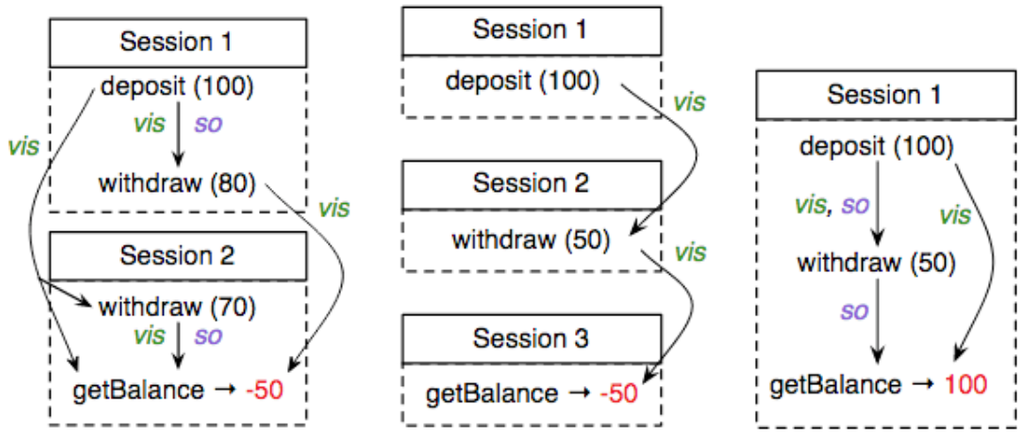
summarization: every now and then (how often?) the history gets boiled down

Q for class: how is summarization done for this particular RDT?

the ubiquitous bank account example

```haskell
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist] -
            sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sell $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

let's play "name that anomaly"



operations

every op gets a *history* of known updates to the object as its first argument

Q for class: what consistency level do each of these three ops need?

summarization: every now and then (how often?) the history gets boiled down

Q for class: how is summarization done for this particular RDT?

the contract language

$$
\begin{array}{llll}
\psi & \in & \texttt{Contract} & ::= \quad \forall(x:\tau).\psi \mid \forall x.\psi \mid \pi \\
\tau & \in & \texttt{EffType} & ::= \quad \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in & \texttt{Prop} & ::= \quad \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & & \mid \quad \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in & \texttt{Relation} & ::= \quad \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \\
 & & & \mid \quad R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

$x, y, \hat{\eta} \in \texttt{EffVar}$ $\qquad$ $\mathsf{Op} \in \texttt{OperName}$

the contract language

a contract is a first-order logic formula
universal quantification over EffVars allowed

$$
\begin{array}{lcll}
\psi & \in & \texttt{Contract} & ::= \quad \forall(x:\tau).\psi \mid \forall x.\psi \mid \pi \\
\tau & \in & \texttt{EffType} & ::= \quad \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in & \texttt{Prop} & ::= \quad \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & & \mid \quad \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in & \texttt{Relation} & ::= \quad \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \\
 & & & \mid \quad R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

$x, y, \hat{\eta} \in \texttt{EffVar}$ $\qquad$ $\mathsf{Op} \in \texttt{OperName}$

the contract language

a contract is a first-order logic formula
universal quantification over EffVars allowed

$$
\begin{array}{lcll}
\psi & \in & \texttt{Contract} & ::= \forall(x:\tau).\psi \mid \forall x.\psi \mid \pi \\
\tau & \in & \texttt{EffType} & ::= \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in & \texttt{Prop} & ::= \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & & \mid \quad \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in & \texttt{Relation} & ::= \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \\
 & & & \mid \quad R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

hey look it's our old friends from Burckhardt et al.

$x, y, \hat{\eta} \in \texttt{EffVar}$ $\qquad$ $\mathsf{Op} \in \texttt{OperName}$

the contract language

$$
\begin{array}{lcll}
\psi & \in & \texttt{Contract} & ::= \quad \forall(x:\tau).\psi \mid \forall x.\psi \mid \pi \\
\tau & \in & \texttt{EffType} & ::= \quad \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in & \texttt{Prop} & ::= \quad \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & & \mid \quad \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in & \texttt{Relation} & ::= \quad \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \\
 & & & \mid \quad R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

$x, y, \hat{\eta} \in \texttt{EffVar}$ $\qquad$ $\mathsf{Op} \in \texttt{OperName}$

a contract is a first-order logic formula
universal quantification over EffVars allowed

hey look it's our old friends from Burckhardt et al.

interesting aside: this is actually a lie
the contract language can't express transitive closure!
(Q for class: why not?)

the contract language

$$
\begin{array}{lcll}
\psi & \in & \texttt{Contract} & ::= \quad \forall(x:\tau).\psi \mid \forall x.\psi \mid \pi \\
\tau & \in & \texttt{EffType} & ::= \quad \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in & \texttt{Prop} & ::= \quad \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & & \quad\ \mid \quad \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in & \texttt{Relation} & ::= \quad \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \\
 & & & \quad\ \mid \quad R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

$$x, y, \hat{\eta} \in \texttt{EffVar} \qquad \mathsf{Op} \in \texttt{OperName}$$

a contract is a first-order logic formula
universal quantification over EffVars allowed

hey look it's our old friends from Burckhardt et al.

interesting aside: this is actually a lie
the contract language can't express transitive closure!
(Q for class: why not?)

some example contracts for operations

($\hat{\eta}$ is the current operation/effect)

for `withdraw`:

$$\forall(a:\texttt{withdraw}).\ \mathsf{sameobj}(a,\hat{\eta}) \Rightarrow a = \hat{\eta} \vee \mathsf{vis}(a,\hat{\eta}) \vee \mathsf{vis}(\hat{\eta},a)$$

for `getBalance`:

$$\forall(a:\texttt{deposit}),(b:\texttt{withdraw}),(c:\texttt{deposit} \vee \texttt{withdraw}).$$
$$(\mathsf{vis}(a,b) \wedge \mathsf{vis}(b,\hat{\eta}) \Rightarrow \mathsf{vis}(a,\hat{\eta}))$$
$$\wedge\ ((\mathsf{so} \cap \mathsf{sameobj})(c,\hat{\eta}) \Rightarrow \mathsf{vis}(c,\hat{\eta}))$$

the contract language

$$
\begin{array}{lcll}
\psi & \in & \texttt{Contract} & ::= \quad \forall(x:\tau).\psi \mid \forall x.\psi \mid \pi \\
\tau & \in & \texttt{EffType} & ::= \quad \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in & \texttt{Prop} & ::= \quad \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & & \mid \quad \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in & \texttt{Relation} & ::= \quad \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \\
 & & & \mid \quad R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

$x, y, \hat{\eta} \in \texttt{EffVar}$ $\qquad$ $\mathsf{Op} \in \texttt{OperName}$

a contract is a first-order logic formula
universal quantification over EffVars allowed

hey look it's our old friends from Burckhardt et al.

interesting aside: this is actually a lie
the contract language can't express transitive closure!
(Q for class: why not?)

some example contracts for operations

($\hat{\eta}$ is the current operation/effect)

for `withdraw`:

$$\forall(a:\texttt{withdraw}).\ \mathsf{sameobj}(a,\hat{\eta}) \Rightarrow a = \hat{\eta} \vee \mathsf{vis}(a,\hat{\eta}) \vee \mathsf{vis}(\hat{\eta},a)$$

for `getBalance`:

$$\forall(a:\texttt{deposit}),(b:\texttt{withdraw}),(c:\texttt{deposit} \vee \texttt{withdraw}).\ (\mathsf{vis}(a,b) \wedge \mathsf{vis}(b,\hat{\eta}) \Rightarrow \mathsf{vis}(a,\hat{\eta})) \wedge ((\mathsf{so} \cap \mathsf{sameobj})(c,\hat{\eta}) \Rightarrow \mathsf{vis}(c,\hat{\eta}))$$

Q for class: what's the contract for `deposit`? why?

stores have contracts, too

eventually consistent store: $\psi_{\mathsf{ec}} = \forall a, b.\ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$

causally consistent store: $\psi_{\mathsf{cc}} = \forall a.\ \mathsf{hbo}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$

strongly consistent store: $\psi_{\mathsf{sc}} = \forall a.\ \mathsf{sameobj}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta}) \ \vee\ \mathsf{vis}(\hat{\eta}, a) \ \vee\ a = \hat{\eta}$

stores have contracts, too

eventually consistent store: $\psi_{\mathsf{ec}} = \forall a, b.\ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$

Q for class:
as definitions of EC go, this one is pretty strong!
what executions does it admit that causal consistency *doesn't*?

causally consistent store: $\psi_{\mathsf{cc}} = \forall a.\ \mathsf{hbo}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$

strongly consistent store: $\psi_{\mathsf{sc}} = \forall a.\ \mathsf{sameobj}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta}) \ \vee\ \mathsf{vis}(\hat{\eta}, a) \ \vee\ a = \hat{\eta}$

stores have contracts, too

eventually consistent store: $\psi_{\mathsf{ec}} = \forall a, b.\ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$

Q for class:
as definitions of EC go, this one is pretty strong!
what executions does it admit that causal consistency *doesn't*?

causally consistent store: $\psi_{\mathsf{cc}} = \forall a.\ \mathsf{hbo}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$

strongly consistent store: $\psi_{\mathsf{sc}} = \forall a.\ \mathsf{sameobj}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta}) \ \vee\ \mathsf{vis}(\hat{\eta}, a) \ \vee\ a = \hat{\eta}$

Qs for class:
does this one remind you of anything from another paper we read recently?
does it correspond to anything in the Viotti and Vukolić zoo?

contract classification

$$\frac{\psi \leq \psi_{\mathsf{sc}}}{\mathsf{WellFormed}(\psi)} \qquad \frac{\psi \leq \psi_{\mathsf{ec}}}{\mathsf{EventuallyConsistent}(\psi)}$$

$$\frac{\psi \not\leq \psi_{\mathsf{ec}} \quad \psi \leq \psi_{\mathsf{cc}}}{\mathsf{CausallyConsistent}(\psi)} \qquad \frac{\psi \not\leq \psi_{\mathsf{cc}} \quad \psi \leq \psi_{\mathsf{sc}}}{\mathsf{StronglyConsistent}(\psi)}$$

note: this really is “classification”,
in the sense that each contract gets classified to *exactly one* consistency level
(i.e., if it’s CausallyConsistent **then it is *not* also** EventuallyConsistent**)**

could also swap in a different set of store consistency levels,
e.g, the four session guarantees,
or even various transaction isolation levels (see paper)

contract classification

$$\frac{\psi \leq \psi_{\mathsf{sc}}}{\mathsf{WellFormed}(\psi)} \qquad \frac{\psi \leq \psi_{\mathsf{ec}}}{\mathsf{EventuallyConsistent}(\psi)}$$

$$\frac{\psi \not\leq \psi_{\mathsf{ec}} \quad \psi \leq \psi_{\mathsf{cc}}}{\mathsf{CausallyConsistent}(\psi)} \qquad \frac{\psi \not\leq \psi_{\mathsf{cc}} \quad \psi \leq \psi_{\mathsf{sc}}}{\mathsf{StronglyConsistent}(\psi)}$$

note: this really is "classification",
in the sense that each contract gets classified to *exactly one* consistency level
(i.e., if it's CausallyConsistent **then it is *not* also** EventuallyConsistent**)**

could also swap in a different set of store consistency levels,
e.g, the four session guarantees,
or even various transaction isolation levels (see paper)

Qs for class:
how would you classify the `getBalance` contract?
how about the `withdraw` contract?
would Owen approve of the lattice in figure 7?

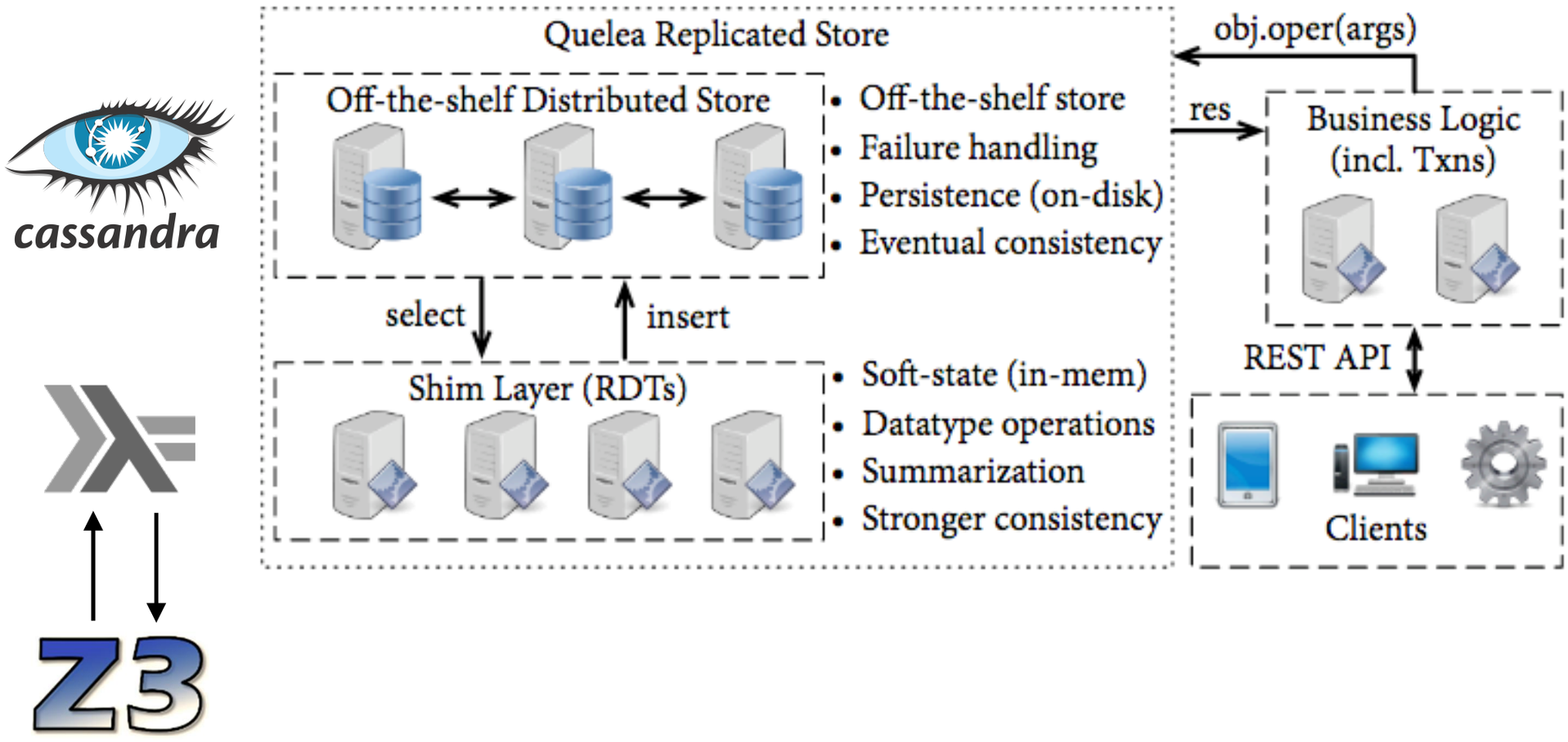
soundness of contract enforcement

Theorem 4 (Soundness of Contract Enforcement). *Let ψ be a well-formed contract of a replicated data type operation op, and let τ denote the consistency class of ψ as determined by the contract classification scheme. For all well-formed execution states* E, E' *such that* $\mathsf{E}, \langle op, \tau \rangle; \sigma \parallel \Sigma \xrightarrow{\eta} \mathsf{E}', \sigma \parallel \Sigma$, *if* $\mathsf{E}' \models \psi_\tau[\eta/\hat{\eta}]$, *then* $\mathsf{E}' \models \psi[\eta/\hat{\eta}]$

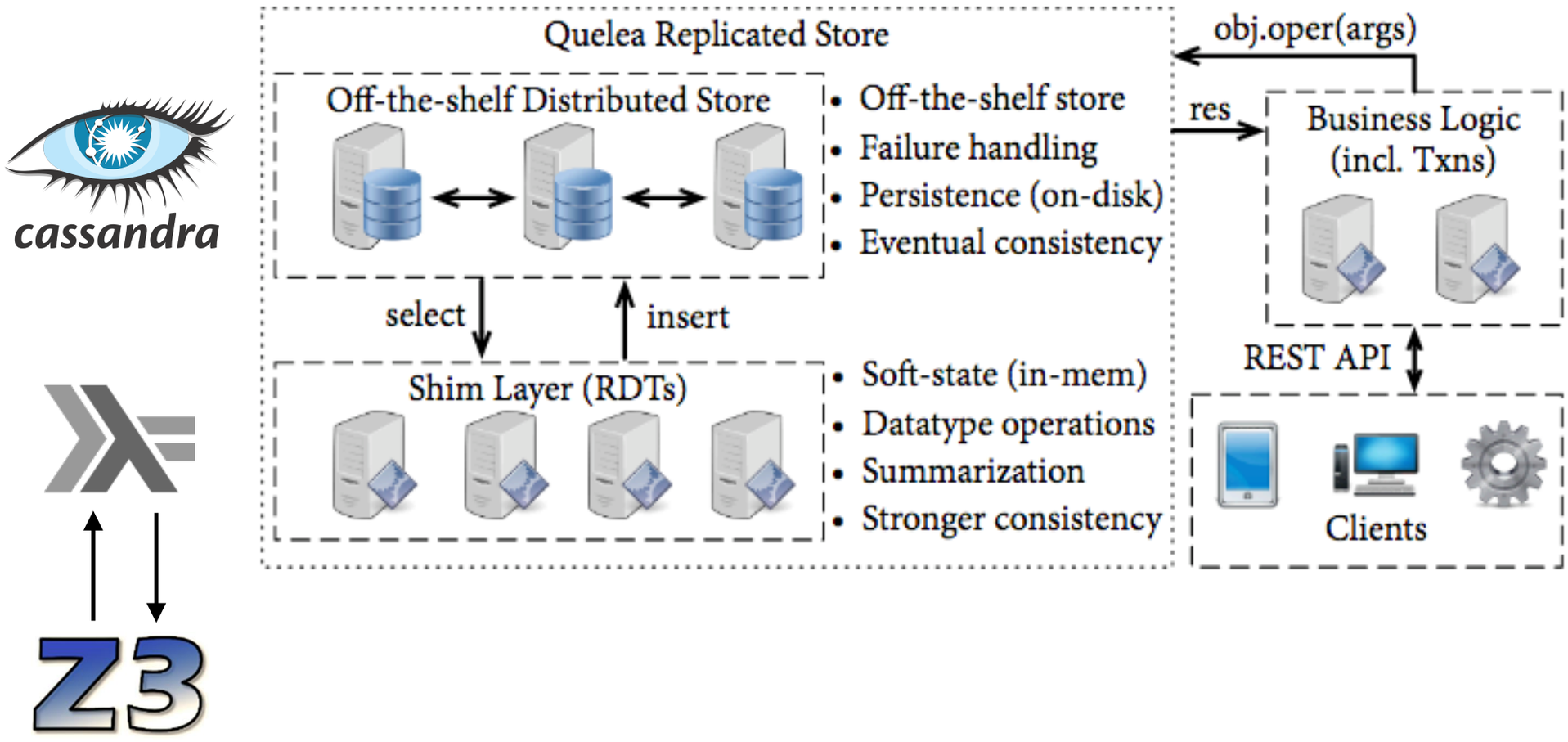
simpler:
“If you take a step and execute an operation
whose contract has been classified in a certain way,
and the resulting execution state satisfies that consistency classification,
then the execution state also satisfies the operation’s original contract.”

or:
“If an operation’s contract is in a particular store consistency class,
and running the operation results in a state satisfying that store consistency,
then the operation’s contract is satisfied.”

implementation

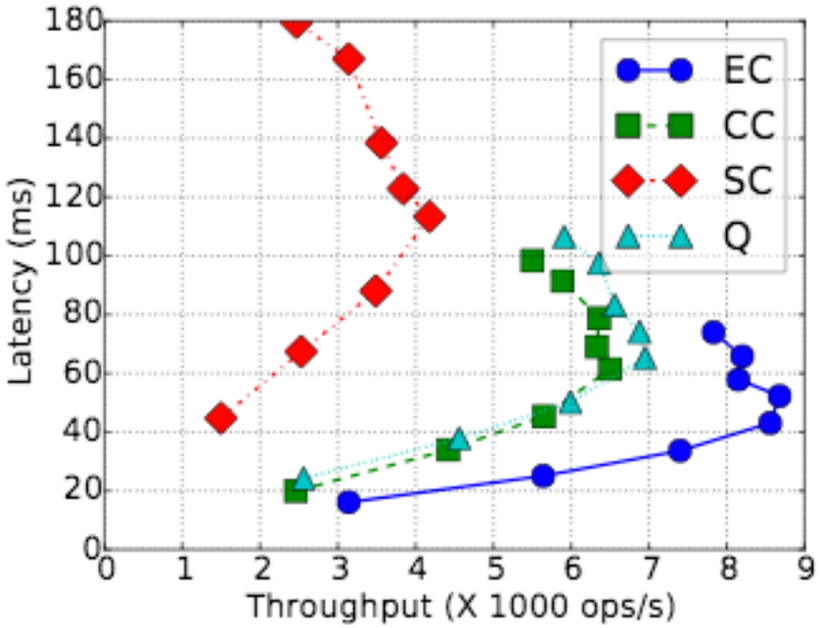


implementation

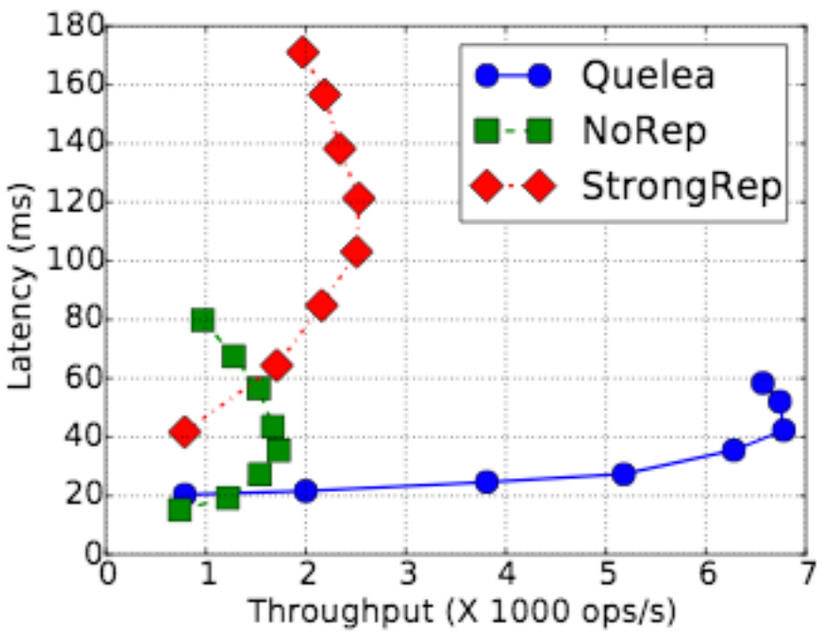


Qs for class:
how can one add causal consistency atop Cassandra's quorum consistency?
(could Holt et al.'s IPA have done the same thing?)

evaluation



nice result:
Quelea does a bit better, throughput-wise, than doing everything at causal consistency (which isn't even "correct")!



Quelea has lower throughput *and* latency than *not replicating*! (why?)

my questions

how often should one summarize?

individual operation contracts can be really fine-grained, only to be classified into more coarse-grained store contract levels. how about a more granular way of interacting with the store?

what do they wish they could've had in the contract language that they couldn't have because of the limitations of SMT?