Sequential Programming for Replicated Data Stores

Nicholas V. Lewchenko\textsuperscript{1} \hspace{1cm} Arjun Radhakrishna\textsuperscript{2}
Akash Gaonkar\textsuperscript{1} \hspace{1cm} Pavol Černý\textsuperscript{1}

\textsuperscript{1}University of Colorado Boulder

\textsuperscript{2}Microsoft

ICFP 2019
Distributed architectures are required for software services that people rely on.
Why Distribute?

Distributed architectures are required for software services that people rely on.

Distributed applications can survive change.

Centralized services cannot.
Why Distribute?

Distributed architectures are required for software services that people rely on.

Distributed applications can survive change.

Centralized services cannot.
Replicated Data Stores

1 1 1 1
Replicated Data Stores

1

+3

deposit 3

1
Replicated Data Stores

Diagram:

- Node 1
- Node 4
- Node 1

Connection:
- Node 4 to Node 1 with an edge labeled +3
Replicated Data Stores

1 4

1 + 4

Failed!
Failed!
Failed!

deposit 4

+4

1

4

4
Replicated Data Stores

Failed!

Availability
Pre- and Post-Condition Reasoning

Given some input to a program, what is its output?
Pre- and Post-Condition Reasoning

Given some input to a program, what is its output?

A formal system: Dependent Refinement Types

\[ \text{sort} : (xs : \text{List}) \rightarrow \{xs' : \text{List} \mid \text{length } xs = \text{length } xs' \} \]
Pre- and Post-Condition Reasoning

Given some input to a program, what is its output?

A formal system: **Dependent Refinement Types**

\[
\text{sort} : (xs : \text{List}) \rightarrow \{xs' : \text{List} \mid \text{length } xs = \text{length } xs'\}
\]

How could we extend this to a replicated store operation?

Conditions on pre-store $\rightarrow$ Conditions on $(\text{Effect } \times \text{Return})$
Pre- and Post-Condition Reasoning

Given some input to a program, what is its output?

A formal system: Dependent Refinement Types

\[ \text{sort} : (xs : \text{List}) \rightarrow \{xs' : \text{List} \mid \text{length } xs = \text{length } xs' \} \]

How could we extend this to a replicated store operation?

Conditions on pre-store \(\rightarrow\) Conditions on \((\text{Effect } \times \text{Return})\)

Consistency
Failed!

4

5

4

3
Failed!

withdraw 3

5

4

5/18
Pre-store withdraw 3

Failed!

Failed!

Pre-stores of withdraw 3:

5 ✓, 4 ✓, 4 ?, 1 ?

Return value

5/18
Failed!

Pre-stores of withdraw 3: 5✓
Failed!

Pre-stores of withdraw 3: 5✓, 4✓
Failed!

Pre-stores of withdraw 3: 5✓, 4✓, 4?
Pre-stores of withdraw 3: \(5\checkmark, 4\checkmark, 4?, 1?\)
So how do we maintain both consistency and availability?
So how do we maintain both consistency and availability?

We don’t…
So how do we maintain both consistency and availability? We don’t…

► **Consistency**: pre/post logic can be enforced
► **Availability**: a called operation always returns a response
► **Partitions**: the network may drop arbitrary messages

**CAP Theorem**: You can only have two.
An Unfortunate Conflict

So how do we maintain both consistency and availability? We don’t…

- **Consistency**: output depends on complete input
- **Availability**: output must eventually be returned
- **Partitions**: complete input might never arrive

**CAP Theorem**: You can only have two.
Partitions are unavoidable for a distributed system.
Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.
Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.

- Consistency: output depends on complete input
Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.

- Consistency: output depends on complete (?) input
Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.

- **Consistency**: output depends on complete (?) input

  
<table>
<thead>
<tr>
<th>Available</th>
<th>Consistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Some def. of “complete” input</td>
</tr>
<tr>
<td></td>
<td>Input must include all pre-stores</td>
</tr>
</tbody>
</table>
But My App Needs All Three!

Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.

- Consistency: output depends on \textit{complete} (?) input

Available \hspace{2cm} Consistent

Empty \hspace{1cm} Some def. of "complete" input \hspace{1cm} Input must include all pre-stores

No balance is universal!
But My App Needs All Three!

Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.

- **Consistency**: output depends on complete (?) input

Available

| Consistency Level |

Consistent

Empty

Input must include all pre-stores

No balance is universal!
The Special Tasks of Replicated Store Programming:
The **Special Tasks** of Replicated Store Programming:

1. Configure segments of application to enforce particular consistency levels.
The **Special Tasks** of Replicated Store Programming:

1. Configure segments of application to enforce particular consistency levels.
2. Verify that chosen consistency levels preserve desired application properties (pre/post).
The Special Tasks of Replicated Store Programming:

0. Invent a domain of useful consistency levels.
1. Configure segments of application to enforce particular consistency levels.
2. Verify that chosen consistency levels preserve desired application properties (pre/post).
Carol is a programming language that simplifies the Special Tasks.
Our Contribution

Carol is a programming language that simplifies the Special Tasks.

- Requires only local, sequential reasoning from the programmer
Our Contribution

Carol is a programming language that simplifies the Special Tasks.

- Requires only local, sequential reasoning from the programmer
- Supports dependent refinement type system (based on Liquid Types)
Carol is a programming language that simplifies the Special Tasks.

- Requires only local, sequential reasoning from the programmer
- Supports dependent refinement type system (based on Liquid Types)

Made possible by a novel replicated store runtime.
Let’s Write an ATM Application!

First, let’s deposit money.

\[
\text{deposit} := \lambda n. \text{issue} (\text{Add } n) \text{ in } n
\]
Let’s Write an ATM Application!

\[
[\text{Add } n] := \lambda x. x + n
\]

First, let’s deposit money.

\[
\text{deposit} := \lambda n. \text{issue (Add } n\text{)} \text{ in } n
\]
Let's Write an ATM Application!

\[
\text{[Add } n\text{]} := \lambda x. x + n
\]

First, let's deposit money.

\[
\text{deposit} := \lambda n. \textbf{issue} (\text{Add } n) \textbf{ in } n
\]

Now we check our balance.
Let’s Write an ATM Application!

\[
[\text{Add } n] := \lambda x. x + n
\]

First, let’s deposit money.

\[
\text{deposit} := \lambda n. \text{issue} (\text{Add } n) \text{ in } n
\]

Now we check our balance.

\[
\text{balance} := \text{query } x \text{ in } x
\]
[Add $n$] := $\lambda x. x + n$

How do we safely withdraw money?
Demanding consistency

$[\text{Add } n] := \lambda x. x + n \quad [\text{Sub } n] := \lambda x. x - n$

How do we safely withdraw money?

\begin{align*}
\text{withdraw} := \lambda n. \text{query } x \text{ in} \\
&\quad \text{if } n \leq x \text{ then} \ (\text{issue } \text{Sub } n \text{ in } n) \ \text{else} \ 0
\end{align*}
Demanding consistency

\[ [\text{Add } n] := \lambda x. x + n \quad [\text{Sub } n] := \lambda x. x - n \]

\[ [x : \text{LEQ}] := x \leq \text{pre-stores} \]

How do we safely withdraw money?

\[
\text{withdraw} := \lambda n. \text{query } x : \text{LEQ} \text{ in } \begin{cases} 
\text{issue Sub } n \text{ in } n & \text{if } n \leq x \text{ then } \\
0 & \text{else}
\end{cases}
\]

A \text{query} term can be annotated with a \text{consistency guard}, which the runtime enforces until termination of the operation.
Demanding consistency

\[ [\text{Add } n] := \lambda x. x + n \quad [\text{Sub } n] := \lambda x. x - n \]

\[ [x : \text{LEQ}] := x \leq \text{pre-stores} \]

How do we safely withdraw money?

\[ \text{withdraw} := \lambda n. \text{query } x : \text{LEQ in} \]
\[ \quad \text{if } n \leq x \text{ then } (\text{issue Sub } n \text{ in } n) \text{ else } 0 \]

A query term can be annotated with a consistency guard, which the runtime enforces until term
Guards: consistency level domain based on data refinements.
Guards: consistency level domain based on data refinements.

1. Provides immediately clear data-based guarantees.
EQV \((x = \text{pre-stores})\)

LEQ

\[\top\]

GEQ \((x \geq \text{pre-stores})\)

Guards: consistency level domain based on data refinements.

1. Provides immediately clear data-based guarantees.
2. Enables local reasoning.
   Meaning of “\(x : \text{LEQ}\)” does not depend on what other operations exist.
Guards: consistency level domain based on data refinements.

1. Provides immediately clear **data-based guarantees**.
2. Enables **local reasoning**.
   Meaning of "x : LEQ" does not depend on what other operations exist.

Special Task 0 ✓
The Carol type system

\[ \Gamma \vdash t : \{ \text{Op } D \ A \mid \varphi_{s,e,r} \} \]

\(D\) is a **Conflict-Aware Replicated Datatype (CARD)** that defines the effects and guards of a store.
\[ \Gamma \vdash t : \{ \textbf{Op} \ D \ A \mid \varphi_{s,e,r} \} \]

\(D\) is a Conflict-Aware Replicated Datatype (CARD) that defines the effects and guards of a store.

\[ \vdash \text{deposit} : (n : \text{Nat}) \rightarrow \{ \textbf{Op} \ Ctr \text{Nat} \mid [e](s) = s + n \} \]
\[ \vdash \text{balance} : \{ \textbf{Op} \ Ctr \text{Int} \mid e = \text{id} \} \]

Everything is an operation!
D is a Conflict-Aware Replicated Datatype (CARD) that defines the effects and guards of a store.

\[ \Gamma \vdash t : \{ \texttt{Op} D A \mid \varphi_{s,e,r} \} \]

\[ \vdash \text{deposit} : (n : \text{Nat}) \rightarrow \{ \texttt{Op} \text{Ctr Nat} \mid \llbracket e \rrbracket(s) = s + n \} \]

\[ \vdash \text{balance} : \{ \texttt{Op} \text{Ctr Int} \mid e = id \} \]

Everything is an operation!

\[ \vdash 5 : \{ \texttt{Op} D \text{Int} \mid e = id \land r = 5 \} \]
Verifying Withdraw

\[ \varphi := (s \geq 0 \Rightarrow \llbracket e \rrbracket(s) \geq 0) \land (r = s - \llbracket e \rrbracket(s)) \]
Verifying Withdraw

\[ \varphi := (s \geq 0 \Rightarrow \mathbb{[e]}(s) \geq 0) \land (r = s - \mathbb{[e]}(s)) \]

1. Account never goes below zero
Verifying Withdraw

ϕ := (s ≥ 0 ⇒ [e](s) ≥ 0) ∧ (r = s − [e](s))

1. Account never goes below zero
2. Value returned to caller is operation’s real effect on store
Verifying Withdraw

\[ \varphi := (s \geq 0 \Rightarrow [e](s) \geq 0) \land (r = s - [e](s)) \]

1. Account never goes below zero
2. Value returned to caller is operation’s real effect on store

\[
\text{withdraw} := \lambda n. \text{query } x : \text{LEQ} \text{ in } \begin{cases} \text{issue Sub } n \text{ in } n \text{ if } n \leq x \text{ then } & \text{else } 0 \end{cases}
\]

\[
\Gamma \vdash \text{LEQ} : \text{Guard(Ctr)} \\
\Gamma, x : \{ \text{Op Ctr Int} \mid r \leq s \} \vdash \text{if } \ldots : \{ \text{Op Ctr Nat} \mid \varphi \} \\
\Gamma \vdash \text{query } x : \text{LEQ} \text{ in if } \ldots : \{ \text{Op Ctr Nat} \mid \varphi \}
\]
Verifying Withdraw

\[ \varphi := (s \geq 0 \Rightarrow \lbrack e \rbrack(s) \geq 0) \land (r = s - \lbrack e \rbrack(s)) \]

1. Account never goes below zero
2. Value returned to caller is operation’s real effect on store

withdraw := λn. query x : LEQ in
            if \( n \leq x \) then (issue Sub n in n) else 0

\[ \Gamma \vdash \text{LEQ : Guard(Ctr)} \]
\[ \Gamma, x : \{ \text{Op Ctr Int} \mid r \leq s \} \vdash \text{if} \ldots : \{ \text{Op Ctr Nat} \mid \varphi \} \]
\[ \Gamma \vdash \text{query x : LEQ in if} \ldots : \]

Special Task 2 ✓
So Who’s Paying For This?

Programmer only needs local, sequential reasoning…
So Who’s Paying For This?

Programmer only needs local, sequential reasoning...

But runtime needs more.
So Who’s Paying For This?

Programmer only needs local, sequential reasoning...

But runtime needs more.

\[
\begin{array}{c}
\text{Add} & \text{Sub} & \text{Set} \\
\text{LEQ} & \text{GEQ} \\
\end{array}
\]

Accords tell the runtime which effects are safe during a query.

Theorem: If \{guard\} is in accord with \{effect\}, then a query using \{guard\} can safely return without including \{effect\}. 
Accords are more reusable and involve less code than full-operation concurrent verification.
What Have We Gained?

Accords are more reusable and involve less code than full-operation concurrent verification.

ATM System
RDT

(balance, deposit, withdraw, countVisitor, moveRobotX, checkRobotRoom)

(conc. ver.)
Accords are **more reusable** and involve **less code** than full-operation concurrent verification.

ATM System vs. Ctr CARD

ATM System

- RDT

(Cconc. ver.)

vs.

Ctr CARD

- deposit
- balance
- withdraw

(Cconc. ver.) (seq. ver.)
What Have We Gained?

Accords are more reusable and involve less code than full-operation concurrent verification.
Future Work: Advanced Runtimes

- Preserving semantics
- Effects or guards—who gets right-of-way?
- Contention management
- Extending semantics/language
- Direct messages for safety-preserving side deals.
Future Work: Advanced Runtimes

Preserving semantics

- Effects or guards—who gets right-of-way?
- Contention management
Future Work: Advanced Runtimes

Preserving semantics

- Effects or guards—who gets right-of-way?
- Contention management

Extending semantics/language

- Direct messages for safety-preserving side deals.
In Conclusion

- Consistency guards isolate programmer from global, concurrent reasoning—operations behave according to local, sequential rules

Haskell DSL and runtime implementation: https://github.com/cuplv/discard
In Conclusion

- **Consistency guards** isolate programmer from global, concurrent reasoning—operations behave according to local, sequential rules
- The local, sequential reasoning is formalized by a dependent refinement type system

Haskell DSL and runtime implementation: https://github.com/cuplv/discard
**In Conclusion**

- **Consistency guards** isolate programmer from global, concurrent reasoning—operations behave according to local, sequential rules.

- The local, sequential reasoning is formalized by a **dependent refinement type** system.

- **Accords** statically capture the concurrent knowledge needed to run many not-yet-written applications.
In Conclusion

▶ **Consistency guards** isolate programmer from global, concurrent reasoning—operations behave according to local, sequential rules

▶ The local, sequential reasoning is formalized by a dependent refinement type system

▶ **Accords** statically capture the concurrent knowledge needed to run many not-yet-written applications

▶ Haskell DSL and runtime implementation: https://github.com/cuplv/discard