Sequential Programming for Replicated Data Stores

Nicholas V. Lewchenko[1] Arjun Radhakrishna[2]
Akash Gaonkar[1] Pavol Černý[1]

[1]University of Colorado Boulder

[2]Microsoft

ICFP 2019

WHY DISTRIBUTE?

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.

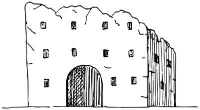
Centralized services cannot.

WHY DISTRIBUTE?

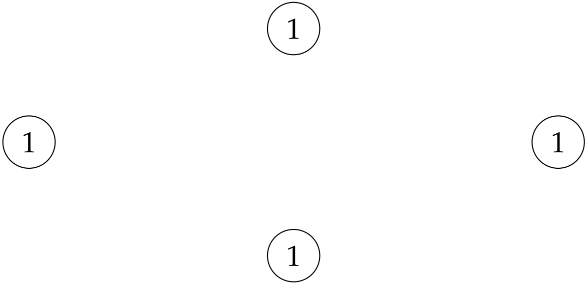
Distributed architectures are required for software services that people rely on.

Distributed applications can survive change.

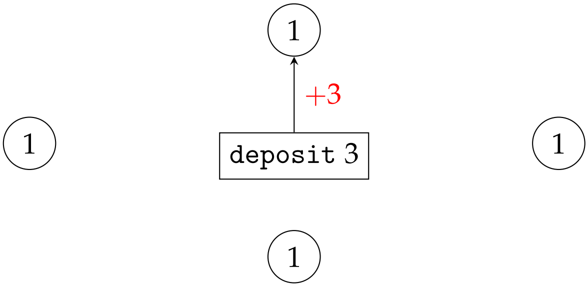
Centralized services cannot.



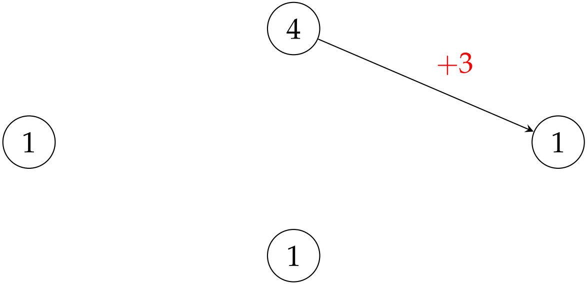
REPLICATED DATA STORES



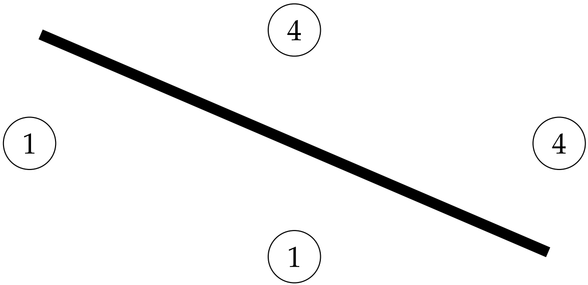
REPLICATED DATA STORES



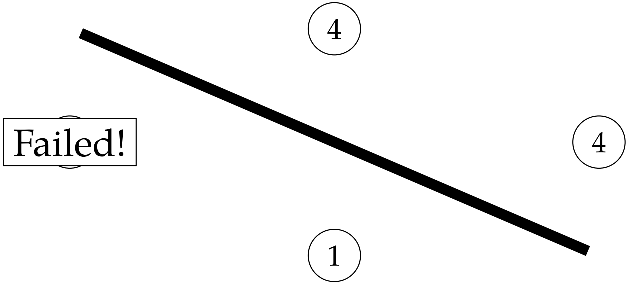
REPLICATED DATA STORES



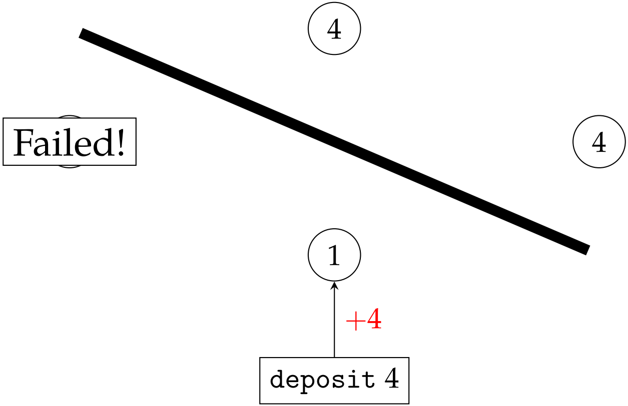
REPLICATED DATA STORES



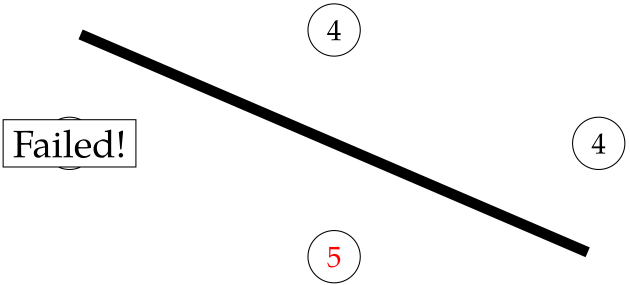
REPLICATED DATA STORES



REPLICATED DATA STORES



REPLICATED DATA STORES



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

$$\texttt{sort} : (xs : \texttt{List}) \to \{xs' : \texttt{List} \mid \texttt{length}\ xs = \texttt{length}\ xs'\}$$

PRE- AND POST-CONDITION REASONING

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

A formal system: Dependent Refinement Types

$$\texttt{sort} : (xs : \texttt{List}) \to \{xs' : \texttt{List} \mid \texttt{length}\ xs = \texttt{length}\ xs'\}$$

How could we extend this to a replicated store operation?

Conditions on pre-store → Conditions on (Effect $\times$ `Return`)

PRE- AND POST-CONDITION REASONING

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

A formal system: Dependent Refinement Types

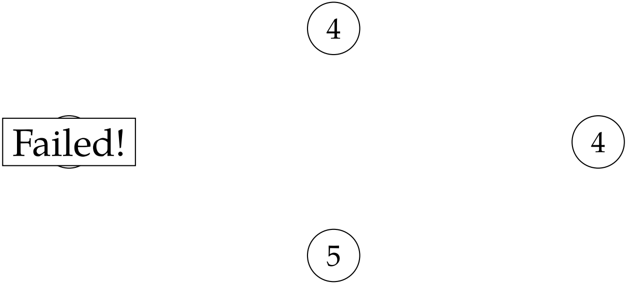
$$\texttt{sort} : (xs : \texttt{List}) \rightarrow \{xs' : \texttt{List} \mid \texttt{length}\ xs = \texttt{length}\ xs'\}$$

How could we extend this to a replicated store operation?

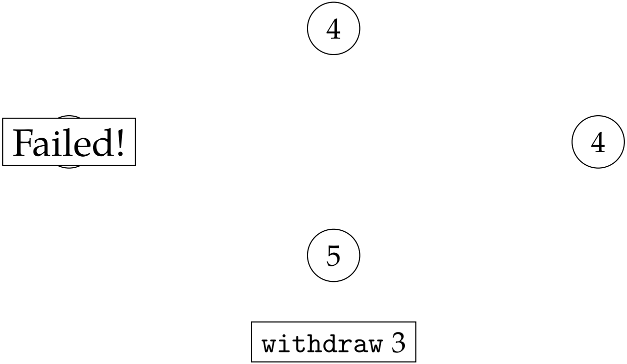
Conditions on pre-store → Conditions on $(\text{Effect} \times \texttt{Return})$

Consistency

REPLICATED DATA STORES



REPLICATED DATA STORES



4

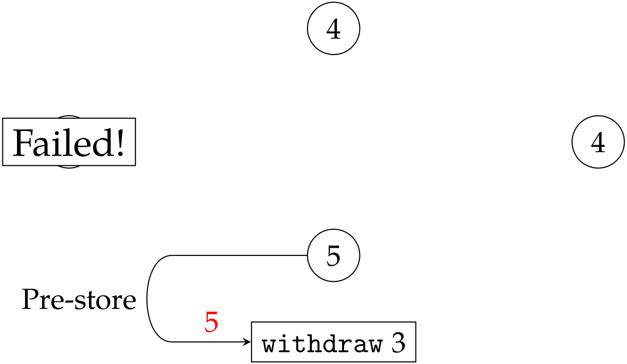
Failed!

4

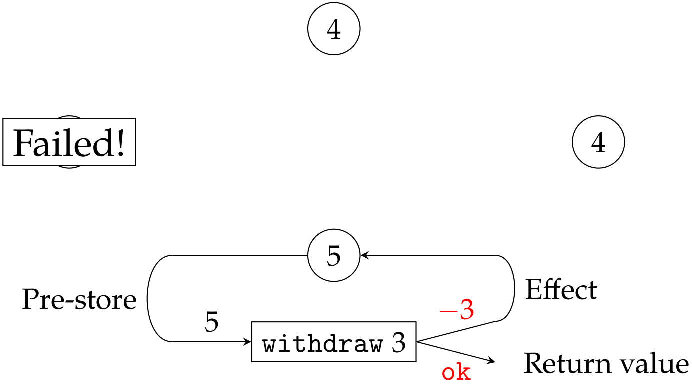
5

withdraw 3

REPLICATED DATA STORES

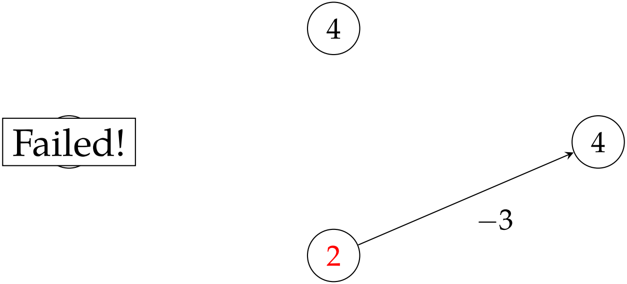


Replicated Data Stores



Pre-stores of `withdraw 3`: 5✓

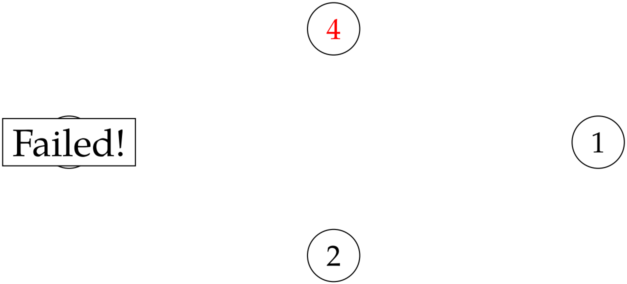
REPLICATED DATA STORES



Also a pre-store of `withdraw 3`

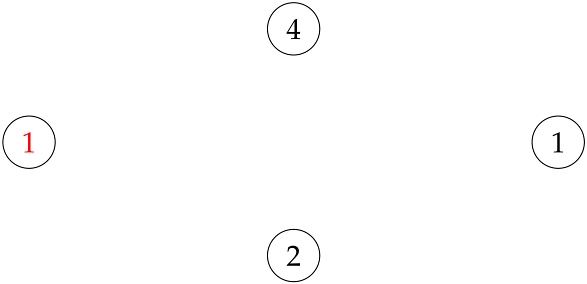
Pre-stores of `withdraw 3`: 5✓, 4✓

REPLICATED DATA STORES



Pre-stores of `withdraw 3`: 5✓, 4✓, 4?

Replicated Data Stores



Pre-stores of `withdraw 3`: 5✓, 4✓, 4?, 1?

AN UNFORTUNATE CONFLICT

So how do we maintain both consistency and availability?

AN UNFORTUNATE CONFLICT

So how do we maintain both consistency and availability?

We don't…

AN UNFORTUNATE CONFLICT

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.

BUT MY APP NEEDS ALL THREE!

Partitions are unavoidable for a distributed system.

BUT MY APP NEEDS ALL THREE!

Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.

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

BUT MY APP NEEDS ALL THREE!

Partitions are unavoidable for a distributed system.

Consistency and Availability can be balanced as needed.

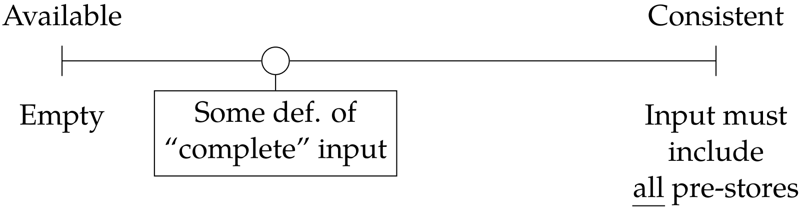
- Consistency: output depends on complete(?) input

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

Consistent

Empty

Some def. of “complete” input

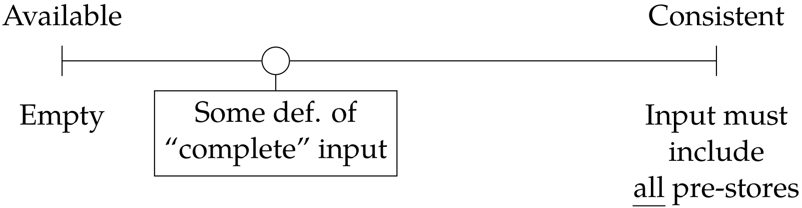
Input must include all pre-stores

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



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

Consistent

Empty

Consistency Level

Input must
include
all pre-stores

No balance is universal!

The <u>Special Tasks</u> 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).

OUR CONTRIBUTION

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)

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)

Made possible by a novel replicated store runtime.

LET'S WRITE AN ATM APPLICATION!

First, let's deposit money.

$$\texttt{deposit} := \lambda n.\ \textbf{issue}\ (\texttt{Add}\ n)\ \textbf{in}\ n$$

LET'S WRITE AN ATM APPLICATION!

$$[\![\texttt{Add}\ n]\!] := \lambda x.\ x + n$$

First, let's deposit money.

$$\texttt{deposit} := \ \lambda n.\ \textbf{issue}\ (\texttt{Add}\ n)\ \textbf{in}\ n$$

LET'S WRITE AN ATM APPLICATION!

$$[\![\texttt{Add}\ n]\!] := \lambda x.\ x + n$$

First, let's deposit money.

$$\texttt{deposit} := \ \lambda n.\ \textbf{issue}\ (\texttt{Add}\ n)\ \textbf{in}\ n$$

Now we check our balance.

LET'S WRITE AN ATM APPLICATION!

$$[\![\texttt{Add}\ n]\!] := \lambda x.\ x + n$$

First, let's deposit money.

$$\texttt{deposit} :=\ \lambda n.\ \textbf{issue}\ (\texttt{Add}\ n)\ \textbf{in}\ n$$

Now we check our balance.

$$\texttt{balance} :=\ \textbf{query}\ x\ \textbf{in}\ x$$

DEMANDING CONSISTENCY

$$[\![\texttt{Add}\ n]\!] := \lambda x.\ x + n$$

How do we safely withdraw money?

DEMANDING CONSISTENCY

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

How do we safely withdraw money?

$$\begin{aligned}\texttt{withdraw} := \lambda n.\ &\textbf{query}\ x\ \textbf{in}\\ &\textbf{if}\ n \leq x\ \textbf{then}\ (\textbf{issue}\ \texttt{Sub}\ n\ \textbf{in}\ n)\ \textbf{else}\ 0\end{aligned}$$

DEMANDING CONSISTENCY

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

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

How do we safely withdraw money?

$$\begin{aligned}\texttt{withdraw} := \lambda n.\ &\textbf{query}\ x : \texttt{LEQ}\ \textbf{in}\\ &\textbf{if}\ n \leq x\ \textbf{then}\ (\textbf{issue}\ \texttt{Sub}\ n\ \textbf{in}\ n)\ \textbf{else}\ 0\end{aligned}$$

A **query** term can be annotated with a consistency guard, which the runtime enforces until termination of the operation.

DEMANDING CONSISTENCY

$$[\![\texttt{Add}\ n]\!] := \lambda x.\ x + n \qquad [\![\texttt{Sub}\ n]\!] := \lambda x.\ x - n$$

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

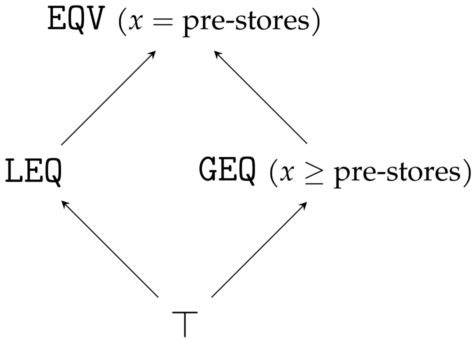
How do we safely withdraw money?

$$\begin{aligned}\texttt{withdraw} := \lambda n.\ &\textbf{query}\ x : \texttt{LEQ}\ \textbf{in}\\ &\textbf{if}\ n \leq x\ \textbf{then}\ (\textbf{issue}\ \texttt{Sub}\ n\ \textbf{in}\ n)\ \textbf{else}\ 0\end{aligned}$$

A **query** term can be annotated with

which the runtime enforces until ter

Special Task 1 ✓



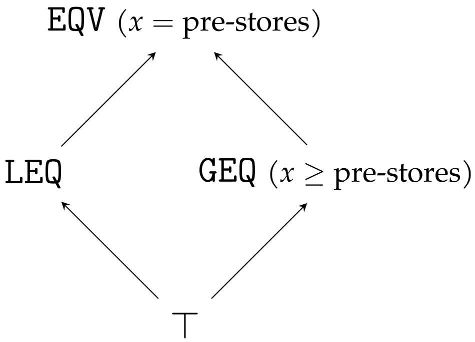
`EQV` ($x =$ pre-stores)

`LEQ`

`GEQ` ($x \geq$ pre-stores)

$\top$

Guards: consistency level domain based on data refinements.



`EQV` (x = pre-stores)

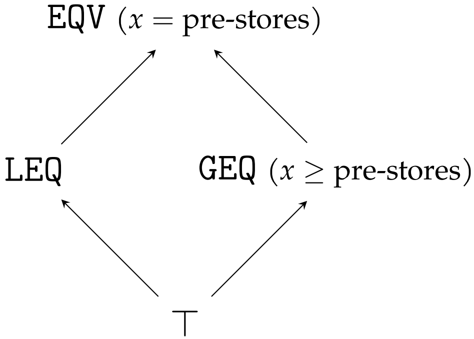
`LEQ`

`GEQ` ($x \geq$ pre-stores)

$\top$

Guards: consistency level domain based on data refinements.

1. Provides immediately clear data-based guarantees.



`EQV` (x = pre-stores)

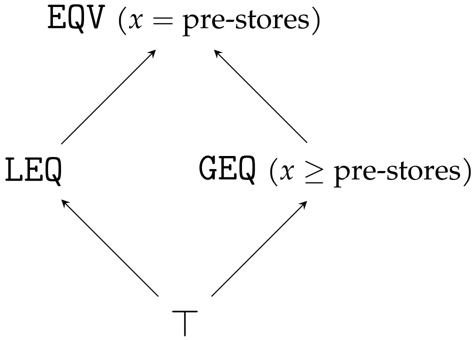
`LEQ`

`GEQ` ($x \geq$ pre-stores)

$\top$

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.



`EQV` (x = pre-stores)

`LEQ`

`GEQ` ($x \geq$ pre-stores)

$\top$

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 d
   what other operations exist.

Special Task 0 ✓

THE CAROL TYPE SYSTEM

$$\boxed{\Gamma \vdash t : \{\ \mathbf{Op}\ D\ A \mid \varphi_{s,e,r}\ \}}$$

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

THE CAROL TYPE SYSTEM

$$\boxed{\Gamma \vdash t : \{\ \mathbf{Op}\ D\ A \mid \varphi_{s,e,r}\ \}}$$

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

$$\begin{aligned} &\vdash \texttt{deposit}\ : (n : \texttt{Nat}) \rightarrow && \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Nat} \mid [\![\mathbf{e}]\!](\mathbf{s}) = \mathbf{s} + n\ \} \\ &\vdash \texttt{balance}\ : && \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Int} \mid \mathbf{e} = \texttt{id}\ \} \end{aligned}$$

THE CAROL TYPE SYSTEM

$$\boxed{\Gamma \vdash t : \{\ \mathbf{Op}\ D\ A \mid \varphi_{s,e,r}\ \}}$$

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

$$\begin{aligned} &\vdash \texttt{deposit} : (n : \texttt{Nat}) \rightarrow && \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Nat} \mid [\![\mathbf{e}]\!](\mathbf{s}) = \mathbf{s} + n\ \} \\ &\vdash \texttt{balance} : && \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Int} \mid \mathbf{e} = \texttt{id}\ \} \end{aligned}$$

Everything is an operation!

$$\vdash 5 : \{\ \mathbf{Op}\ D\ \texttt{Int} \mid \mathbf{e} = \texttt{id} \wedge \mathbf{r} = 5\ \}$$

VERIFYING WITHDRAW

$$\varphi := (\mathbf{s} \geq 0 \Rightarrow [\![\mathbf{e}]\!](\mathbf{s}) \geq 0) \wedge (\mathbf{r} = \mathbf{s} - [\![\mathbf{e}]\!](\mathbf{s}))$$

VERIFYING WITHDRAW

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

1. Account never goes below zero

VERIFYING WITHDRAW

$$\varphi := (\mathbf{s} \geq 0 \Rightarrow [\![\mathbf{e}]\!](\mathbf{s}) \geq 0) \wedge (\mathbf{r} = \mathbf{s} - [\![\mathbf{e}]\!](\mathbf{s}))$$

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

VERIFYING WITHDRAW

$$\varphi := (\mathbf{s} \geq 0 \Rightarrow [\![\mathbf{e}]\!](\mathbf{s}) \geq 0) \wedge (\mathbf{r} = \mathbf{s} - [\![\mathbf{e}]\!](\mathbf{s}))$$

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

$$\begin{aligned} \texttt{withdraw} := \lambda n.\ & \textbf{query}\ x : \texttt{LEQ}\ \textbf{in} \\ & \textbf{if}\ n \leq x\ \textbf{then}\ (\textbf{issue}\ \texttt{Sub}\ n\ \textbf{in}\ n)\ \textbf{else}\ 0 \end{aligned}$$

$$\frac{\Gamma \vdash \texttt{LEQ} : \mathrm{Guard}(\texttt{Ctr}) \qquad \Gamma, x : \{\ \textbf{Op}\ \texttt{Ctr}\ \texttt{Int} \mid \mathbf{r} \leq \mathbf{s}\ \} \vdash \textbf{if} \ldots : \{\ \textbf{Op}\ \texttt{Ctr}\ \texttt{Nat} \mid \varphi\ \}}{\Gamma \vdash \textbf{query}\ x : \texttt{LEQ}\ \textbf{in}\ \textbf{if} \ldots : \{\ \textbf{Op}\ \texttt{Ctr}\ \texttt{Nat} \mid \varphi\ \}}$$

VERIFYING WITHDRAW

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

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

$$\begin{aligned} \texttt{withdraw} := \lambda n.\ & \mathbf{query}\ x : \texttt{LEQ}\ \mathbf{in} \\ & \mathbf{if}\ n \leq x\ \mathbf{then}\ (\mathbf{issue}\ \texttt{Sub}\ n\ \mathbf{in}\ n)\ \mathbf{else}\ 0 \end{aligned}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{LEQ} : \mathrm{Guard}(\texttt{Ctr}) \\ \Gamma, x : \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Int} \mid \mathbf{r} \leq \mathbf{s}\ \} \vdash \mathbf{if} \ldots : \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Nat} \mid \varphi\ \}\end{array}}{\Gamma \vdash \mathbf{query}\ x : \texttt{LEQ}\ \mathbf{in}\ \mathbf{if} \ldots :}$$

Special Task 2 ✓

SO WHO'S PAYING FOR THIS?

Programmer only needs local, sequential reasoning…

SO WHO'S PAYING FOR THIS?

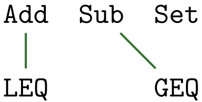
Programmer only needs local, sequential reasoning…

But runtime needs more.

SO WHO'S PAYING FOR THIS?

Programmer only needs local, sequential reasoning…

But runtime needs more.



`Add` `Sub` `Set`

`LEQ` `GEQ`

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

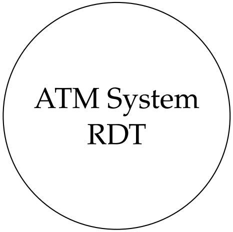
Theorem: If {guard} is in accord with {effect}, then a query using {guard} can safely return without including {effect}.

WHAT HAVE WE GAINED?

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

WHAT HAVE WE GAINED?

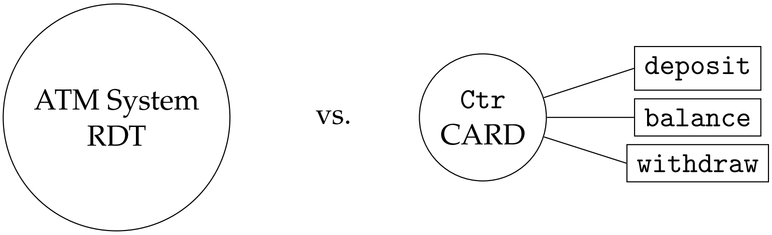
Accords are more reusable and involve less code than full-operation concurrent verification.



(conc. ver.)

WHAT HAVE WE GAINED?

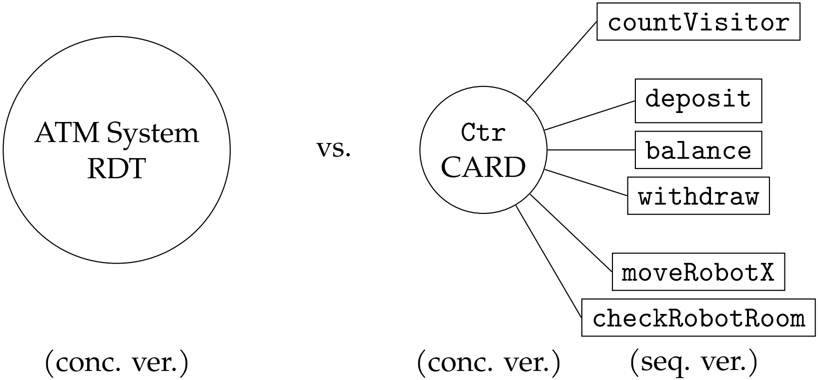
Accords are more reusable and involve less code than full-operation concurrent verification.



(conc. ver.) (conc. ver.) (seq. ver.)

WHAT HAVE WE GAINED?

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



FUTURE WORK: ADVANCED RUNTIMES

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

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

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`