Reasoning and Programming with Commutativity

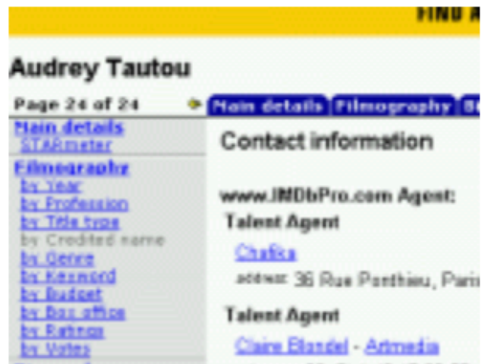
Eric Koskinen
Stevens Institute of Technology
www.erickoskinen.com

MIT • Monday, May 9, 2022

The Internet Movie Database

Visited by over 13 million movie lovers each month!

IMDbPro.com, the new website for the entertainment industry, now features over 22,000 agent/contact listings. Click here for a free trial.



Contact Information for Over 22,000 Professionals on IMDbPro.com

If you're looking for agent contact information for industry professionals, IMDbPro.com has contact data for over 22,000 names and the information is being added to and updated daily. If you're looking for company contact information, international, domestic and daily box-office numbers, IMDbPro.com is the site for you. If you're looking for news and film reviews from the *Hollywood Reporter*, IMDbPro.com is where you need to go. Try IMDbPro.com for free for two weeks. Click here for details.

Today's IMDb Poll Question Is:

Daily Poll It bothers me the most that.... (Vote for your favorite actors and films of 2002 with our new and improved 2002 poll.) (vote)

SAG Nominees in Road to the Oscars

It's getting hotter and hotter in the Windy City, as the razzle-dazzle musical *Chicago* made off with five nominations from the Screen Actors Guild Awards, including Best Performance by a Cast in a Motion Picture. Read more about the SAG awards in our Road to the Oscars® section. You'll also find other awards news and Oscar® trivia and quotes from Oscar®-nominated films. Best of all, have your say and cast your ballot in our new and improved Best of 2002 Poll. It's all on

Search the database for

All | Go!

More searches | Tips

IMDbPro.com free trial

Tops at the Box Office

1. Darkness Falls
2. Kangaroo Jack
3. Chicago
4. National Security
5. Just Married

more

Opening this Week

- The Recruit
- Final Destination
- Biker Boyz
- Lost in La Mancha
- May

more

Coming Soon

- How to Lose a Guy
- Shanghai Knights
- Deliver Us from Eva
- Daredevil
- The Jungle Book 2
- Max
- The Life of David Gale

Movie and TV News

Wed January 29, 2003:

Celebrity News

- Townshend: Email Will Clear Me
- Britney Dumps Durst for Farrell
- Crowe to Miss BAFTAs

Studio Briefing

- SAG Nods Go to 'Chicago,' 'The Hours'
- Eisner Getting $5-Million Stock Bonus
- O'Toole Rejects Oscar; Academy Says He Earned It

Celebrity Interviews/Articles

- *Adaptation* Filmmakers
- Glen Keane - *Treasure Planet*

Cool Feature!

Happy Birthday to:

Thursday, January 30, 2003:

- Christian Bale (29)
- Gene Hackman (73)
- Wilmer Valderrama (23)
- Vanessa Redgrave (66)

more birthdays

Cool Services

- Daily Newsletter
- Contact IMDb.com

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

IMDb®

Can we serve web pages faster?

Can we serve web pages faster?

thttpd - tiny/turbo/throttling HTTP server

Fetch the software. Release notes.

thttpd is a simple, small, portable, fast, and secure HTTP server.

Simple:
: It handles only the minimum necessary to implement HTTP/1.1. Well, maybe a little more than the minimum.

Small:
: See the comparison chart. It also has a very small run-time size, since it does not fork and is very careful about memory allocation.

Portable:
: It compiles cleanly on most any Unix-like OS, specifically including FreeBSD, SunOS 4, Solaris 2, BSD/OS, Linux, OSF.

Fast:
: In typical use it's about as fast as the best full-featured servers (Apache, NCSA, Netscape). Under extreme load it's much faster.

Secure:
: It goes to great lengths to protect the web server machine against attacks and breakins from other sites.

It also has one extremely useful feature (URL-traffic-based throttling) that no other server currently has. Plus, it supports IPv6 out of the box, no patching required.

More specific info:

- HTMLized man page
- thttpd notes

Let me try to write something faster myself.

Wow, concurrency is hard!

Why does writing concurrent programs have to be so hard?

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects

Maurice Herlihy   Eric Koskinen

Computer Science Department, Brown University

{mph,ejk}@cs.brown.edu

Abstract

We describe a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. As long as the linearizable implementation satisfies certain regularity properties (informally, that every method has an inverse), we define a simple wrapper for the linearizable implementation that guarantees that concurrent transactions without inherent conflicts can synchronize at the same granularity as the original linearizable implementation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features – Frameworks; Concurrent programming structures; E.1 [*Data Structures*]: Distributed data structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages, Theory

Synchronizing via read/write conflicts has one substantial advantage: it can be done automatically without programmer participation. It also has a substantial disadvantage: it can severely and unnecessarily restrict concurrency for certain shared objects. If these objects are subject to high levels of contention (that is, they are "hot-spots"), then the performance of the system as a whole may suffer.

Here is a simple example. Consider a mutable set of integers that provides add(x), remove(x) and contains(x) methods with the obvious meanings. Suppose we implement the set as a sorted linked list in the usual way. Each list node has two fields, an integer value and a node reference next. List nodes are sorted by value, and values are not duplicated. Integer x is in the set if and only if a list node has value field x. The add(x) method reads along the list until it encounters the largest value less than x. Assuming x is absent, it creates a node to hold x, and lin

Consider a set whose state is $\{1, 3, 5\}$
to add 2 to the set and transaction B is abou

PPoPP 2008

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects

Coarse-Grained Transactions

Eric Koskinen
University of Cambridge

Matthew Parkinson
University of Cambridge

Maurice Herlihy
Brown University

POPL 2010

The Push/Pull Model of Transactions

Eric Koskinen *
IBM TJ Watson Research Center, USA

Matthew Parkinson
Microsoft Research Cambridge, U

PLDI 2015

Adding Concurrency to Smart Contracts

Thomas Dickerson
Brown University
thomas_dickerson@brown.edu

Paul Gazzillo
Yale University
paul.gazzillo@yale.edu

Maurice Herlihy

Eric Koskinen

PODC 2017

Still have to write concurrent programs.

Oh the dream of parallelizing compilers!

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Still have to write concurrent programs.

Oh the dream of parallelizing compilers!

Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers

MARTIN C. RINARD
Massachusetts Institute of Technology
and
PEDRO C. DINIZ
University of Southern California / Information Sciences Institute

---

This article presents a new analysis technique, commutativity analysis, for automatically parallelizing computations that manipulate dynamic, pointer-based data structures. Commutativity analysis views the computation as composed of operations on objects. It then analyzes the pro-

Stevens Institute of Technology
The Innovation University®

Wanted to understand program analysis better…

- Moved to Cambridge, UK
- Dissertation on **program analysis for temporal logic** verification
- Abstraction-refinement, automata, …
- Started to think more and more about how

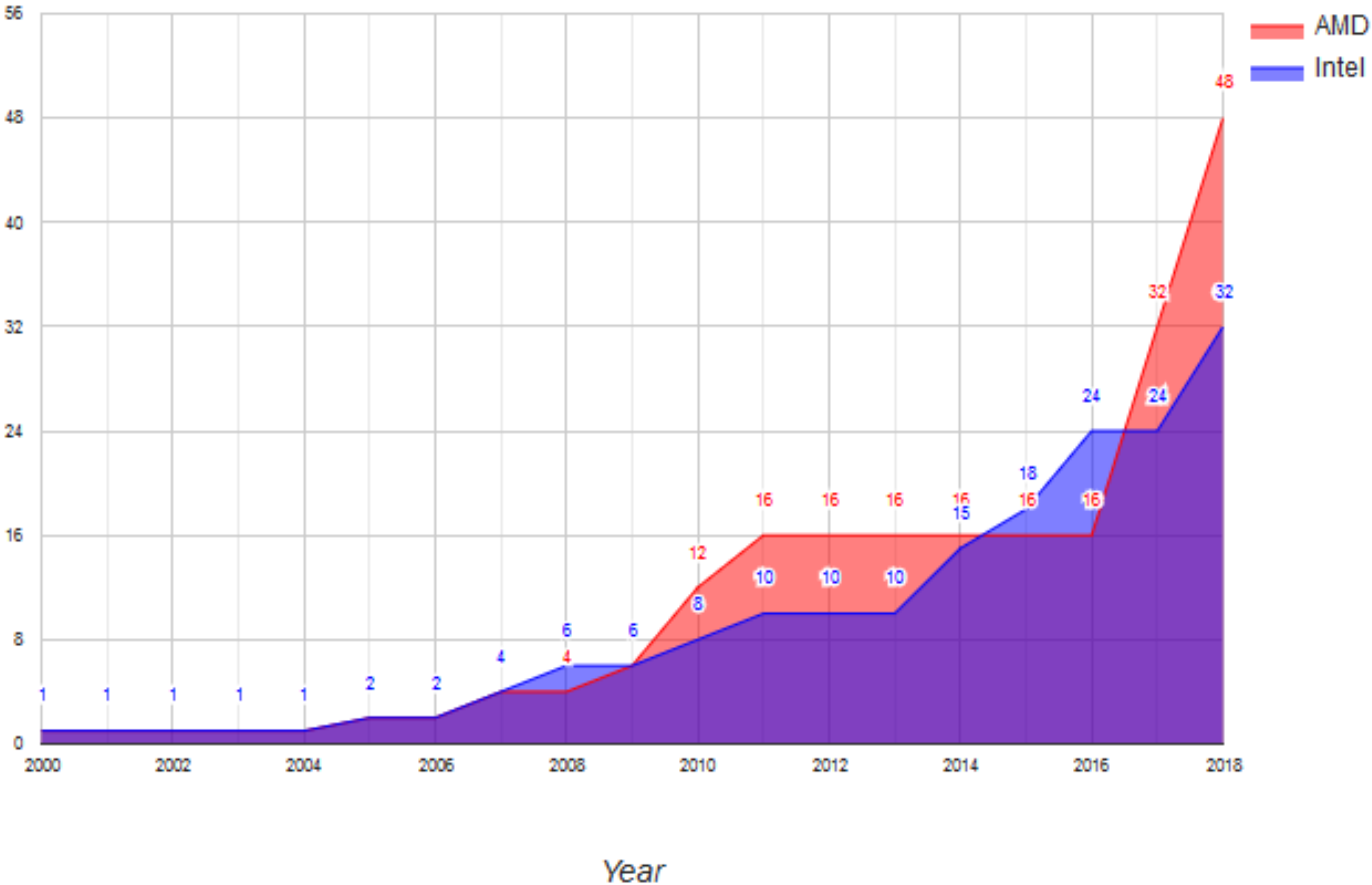
STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Wanted to understand program analysis better…

- Moved to Cambridge, UK
- Dissertation on **program analysis for temporal logic** verification
- Abstraction-refinement, automata, …
- Started to think more and more about how

Symbolic program analysis could enable **parallelization**

Highest amount of cores per CPU (AMD vs Intel year by year)



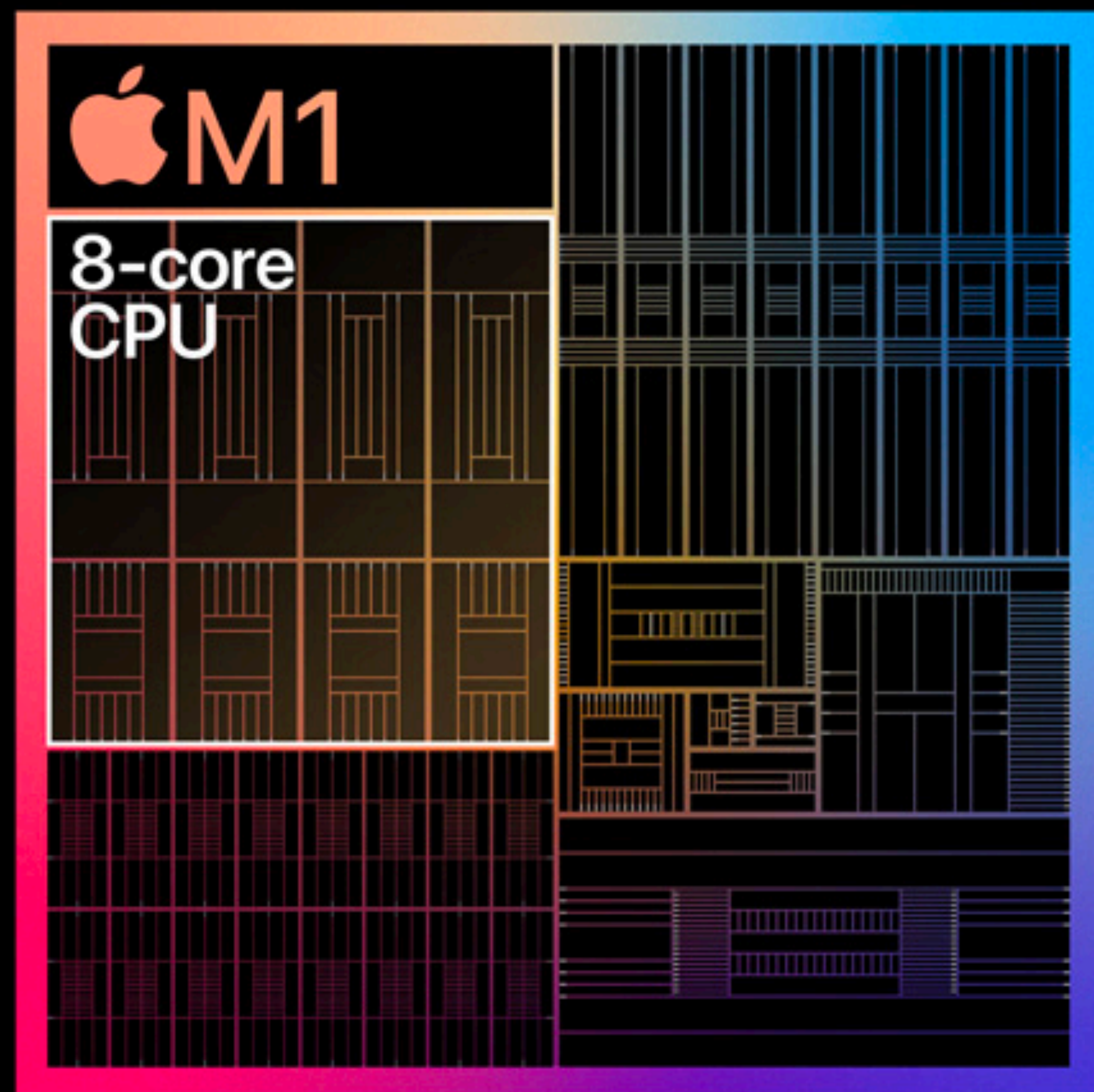
8-core CPU

The highest-performance CPU
we've ever built.

Up to

3.5x

faster CPU
performance[1]



Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

http://jakascorner.com/blog/2016/06/omp-for-reduction.html

Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

Exploits Commutativity

http://jakascorner.com/blog/2016/06/omp-for-reduction.html

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

***Commutativity** is a well-known strategy for concurrency.*

- **Databases**. e.g. Weihl 1988.
- **Parallelizing compilers**. e.g. Rinard & Diniz, TOPLAS 1997.
- **Parallel graph algorithms**. e.g. Kulkarni et al, PLDI 2007.
- **Transactional memory**. e.g. Ni et al., PPoPP 2007. Herlihy & Koskinen, PPoPP 2008. Bronson et al., PODC 2010. Hassan et al., PPoPP 2014. Koskinen & Parkinson, PLDI 2015. Dickerson et al., APLAS 2019.
- **Runtime systems**. e.g. Tripp et al, OOPSLA 2011.
- **Software scalability**. e.g. Clements et al., TOCS 2015.
- **Layered concurrent programs**. e.g. Kragl & Qadeer, CAV 2018.

http://jakascorner.com/blog/2016/06/omp-for-reduction.html

Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

***Commutativity** is a well-known strategy for concurrency.*

. Weihl 1988.

ompilers. e.g. Rinard & Diniz, TOPLAS 1997.

algorithms. e.g. Kulkarni et al, PLDI 2007.

- **Transactional memory**. e.g. Ni et al., PPoPP 2007. Herlihy & Koskinen, PPoPP 2008. Bronson et al., PODC 2010. Hassan et al., PPoPP 2014. Koskinen & Parkinson, PLDI 2015. Dickerson et al., APLAS 2019.
- **Runtime systems**. e.g. Tripp et al, OOPSLA 2011.
- **Software scalability**. e.g. Clements et al., TOCS 2015.
- **Layered concurrent programs**. e.g. Kragl & Qadeer, CAV 2018.

Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

***Commutativity** is a well-known strategy for concurrency.*

. Weihl 1988.

ompilers. e

algorithm

- **Transactional memory**. e
  Bronson et al., PODC 2010. Hass
  Dickerson et al., APLAS 2019.
- **Runtime systems**. e.g. Trip
- **Software scalability**. e.g. C
- **Layered concurrent prog**

Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers

MARTIN C. RINARD
Massachusetts Institute of Technology
and
PEDRO C. DINIZ
University of Southern California / Information Sciences Institute

This article presents a new analysis technique, commutativity analysis, for automatically parallelizing computations that manipulate dynamic, pointer-based data structures. Commutativity analysis views the computation as composed of operations on objects. It then analyzes the program at this granularity to discover when operations commute (i.e., generate the same final result

Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

***Commutativity** is a well-known strategy for concurrency.*

. Weihl 1988.

ompilers. e

algorithm

- **Transactional memory**. e

Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers

MARTIN C. RINARD

of Technology

California / Information Sciences Institute

analysis technique, commutativity analysis, for automatically paral-
manipulate dynamic, pointer-based data structures. Commutativity
tion as composed of operations on objects. It then analyzes the pro-
discover when operations commute (i.e., generate the same final result

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

AUSTIN T. CLEMENTS, M. FRANS KAASHOEK, NICKOLAI ZELDOVICH, and ROBERT T. MORRIS, MIT CSAIL
EDDIE KOHLER, Harvard University

What opportunities for multicore scalability are latent in software interfaces, such as system call APIs? Can scalability challenges and opportunities be identified even before any implementation exists, simply by considering interface specifications? To answer these questions, we introduce the scalable commutativity rule: *whenever interface operations commute, they can be implemented in a way that scales.* This rule is useful throughout the development process for scalable multicore software, from the interface design through

Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

***Commutativity** is a well-known strategy for concurrency.*

. Weihl 1988.

ompilers

- **Transac**

nalysis Technique

s Institute

y analysis, for automatically paral-
d data structures. Commutativity
objects. It then analyzes the pro-
(i.e., generate the same final result

The Scalable Comm
for Multicore Proces

AUSTIN T. CLEMENTS, M
and ROBERT T. MORRIS
EDDIE KOHLER, Harvard

What opportunities for multi
Can scalability challenges an
by considering interface specif
rule: *whenever interface opera*
useful throughout the developm

A Revised and Verified Proof of the Scalable Commutativity Rule

Lillian Tsai[†], Eddie Kohler[*], M. Frans Kaashoek[†], and Nickolai Zeldovich[†]
[†] MIT CSAIL   [*] Harvard University

1 Introduction

This paper explains a flaw in the published proof of the Scalable Commutativity Rule (SCR) [1], presents a revised and formally verified proof of the SCR in the Coq proof assistant, and discusses the insights and open questions raised from our experience proving the SCR.

2 The Scalable Commutativity Rule

In order to explore the connection between commutativity and scalability in practical systems, Clements et al. [1] defined a new type of commutativity called *SIM commutativity*,[1] a property that can hold of certain interface specifications. This was used to state and prove the Scalable Commutativity Rule (SCR), which claims that every SIM-

A *specification* models an interface's behavior as a prefix-closed set of well-formed histories. A system execution is "correct" according to the specification if its trace is included in the specification. For instance, if $\mathscr{S}$ corresponded to the POSIX specification, then $[\textbf{getpid}_\alpha, \overline{\textbf{92}}_\alpha] \in \mathscr{S}$ (a process may have PID 92) but $[\textbf{getpid}_\alpha, \overline{\textbf{ENOENT}}_\alpha] \notin \mathscr{S}$ (the **getpid**() system call may not return that error). A specification constrains both invocations and responses: $[\textbf{NtAddAtom}_\alpha]$ is not in the POSIX specification because **NtAddAtom** is not a POSIX system call.

An *implementation* is an abstract machine that takes invocations and calculates responses. The original proof of the SCR by Clements et al. [1] (also presented in Section 2.4) uses a class of machines on which conflict-freedom

Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

***Commutativity** is a well-known strategy for concurrency.*

. Weihl 1988.

ompilers

nalysis Technique

- Transac

The Scalable Comm
for Multicore Proce

AUSTIN T. CLEMENTS,
and ROBERT T. MORRIS
EDDIE KOHLER, Harvard

What opportunities for multi
Can scalability challenges an
by considering interface specif
rule: *whenever interface opera*
useful throughout the developm

A Revised and Verified Proof of the Scalable Commutativity Rule

Lillian Tsai[†], Eddie Kohler[*], M. Frans Kaashoek[†], and Nickolai Zeldovich[†]
[†] MIT CSAIL [*] Harvard University

1 Introduction

This paper explains
able Commutativity
formally verified pro
and discusses the ins
experience proving t

2 The Scalable

In order to explore
and scalability in pr
fined a new type o
tivity,[1] a property t
ifications. This was
Commutativity Rule

ScaleFS: A Multicore-Scalable File System

by

Rasha Eqbal

B.Tech.(Hons.) in Computer Science and Engineering
Indian Institute of Technology Kharagpur (2011)

ally paral-
mutativity
es the pro-
final result

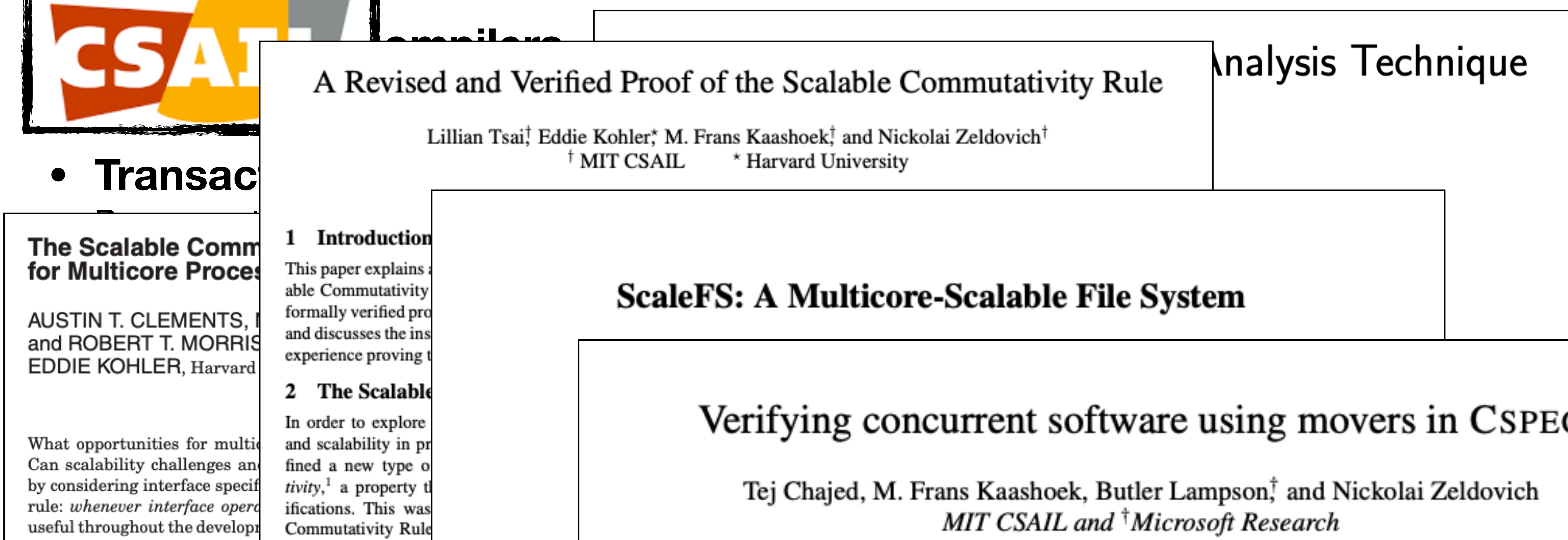
Automatic parallelization

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 10; i++)
{
    sum += a[i];
}
```

Commutativity *is a well-known strategy for concurrency.*

. Weihl 1988.

- **Transac**



Analysis Technique

A Revised and Verified Proof of the Scalable Commutativity Rule

Lillian Tsai[†], Eddie Kohler[*], M. Frans Kaashoek[†], and Nickolai Zeldovich[†]
[†] MIT CSAIL [*] Harvard University

The Scalable Comm
for Multicore Proces

AUSTIN T. CLEMENTS,
and ROBERT T. MORRIS
EDDIE KOHLER, Harvard

What opportunities for multi
Can scalability challenges an
by considering interface specif
rule: *whenever interface opera*
useful throughout the developm

1 Introduction

This paper explains
able Commutativity
formally verified pro
and discusses the ins
experience proving t

2 The Scalable

In order to explore
and scalability in pr
fined a new type o
tivity,[1] a property t
ifications. This was
Commutativity Rule

ScaleFS: A Multicore-Scalable File System

Verifying concurrent software using movers in CSPEC

Tej Chajed, M. Frans Kaashoek, Butler Lampson[†], and Nickolai Zeldovich
MIT CSAIL and [†]*Microsoft Research*

Automatic parallelization

But programs don't always commute …

```
x = calc1(a);
c = c + (x*x);
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

```
x = calc1(a);
c = c + (x*x);
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Expensive, pure computation

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

```
x = calc1(a);
c = c + (x*x);
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Expensive, pure computation

Another expensive, pure computation

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

```
x = calc1(a);
c = c + (x*x);
--------------------------
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Expensive, pure computation

Another expensive, pure computation

Can we run them in parallel?

Let’s try separating splitting the code.

Wouldn’t it be nice if we could parallelize these two blocks?

```
x = calc1(a);
c = c + (x*x);
```

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Can we run them in parallel?

```
x = calc1(a);
c = c + (x*x);
```

Observation on c

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Can we run them in parallel?

Mutation of c

```
x = calc1(a);
c = c + (x*x);
```

Observation on c

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Can we run them in parallel?

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Mutation of c

```
x = calc1(a);
c = c + (x*x);
```

Observation on c

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Can we run them in parallel?

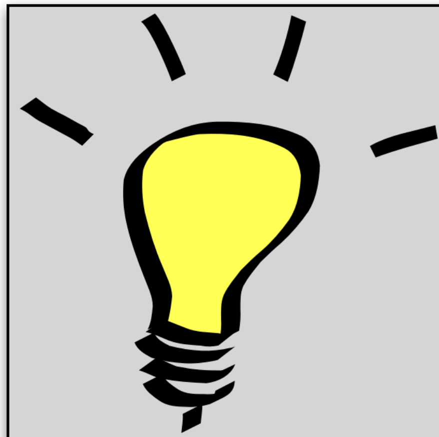
A simple dataflow analysis cannot parallelize them.

Dataflow dependency prevents naive parallelization.

Splitting differently doesn't help; x conflicts.

```
x = calc1(a);
c = c + (x*x);
```

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```



Idea: Making commutativity *explicit* can allow us to **weaken** the dataflow dependency.

Consider: what if **`c>0`** initially?

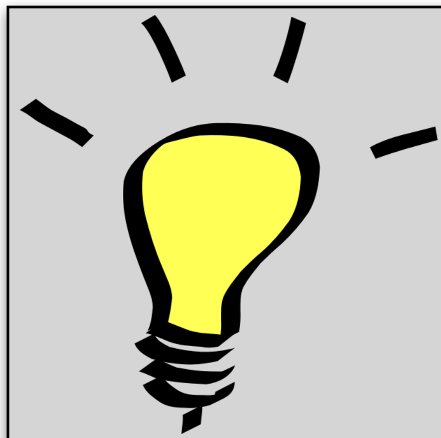
Then these blocks are semantically **independent**.

(*with some atomicity assumptions*)

```
x = calc1(a);
c = c + (x*x);
```

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Observes if c>0
Decreases c by 1



Idea: Making commutativity *explicit* can allow us to **weaken** the dataflow dependency.

Consider: what if **c>0** initially?

Then these blocks are semantically **independent**.

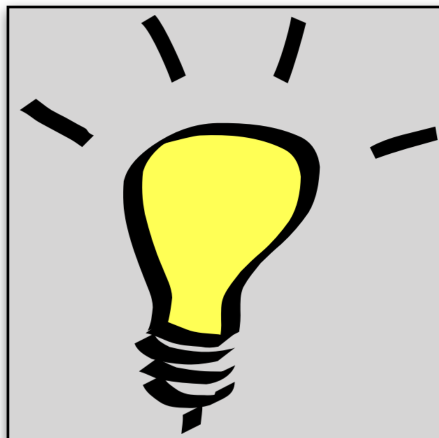
(*with some atomicity assumptions*)

Increases c

```
x = calc1(a);
c = c + (x*x);
```

Observes if c>0
Decreases c by 1

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```



Idea: Making commutativity *explicit* can allow us to **weaken** the dataflow dependency.

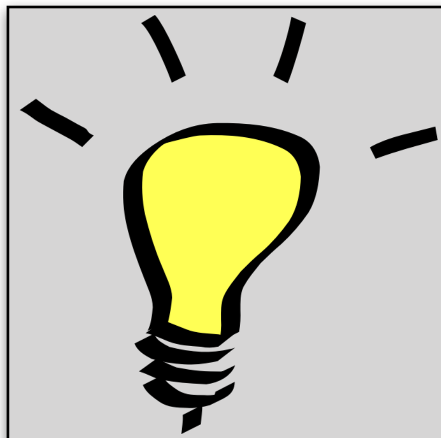
Consider: what if **`c>0`** initially?

Then these blocks are semantically **independent**.

(*with some atomicity assumptions*)

```
x = calc1(a);
c = c + (x*x);
```

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

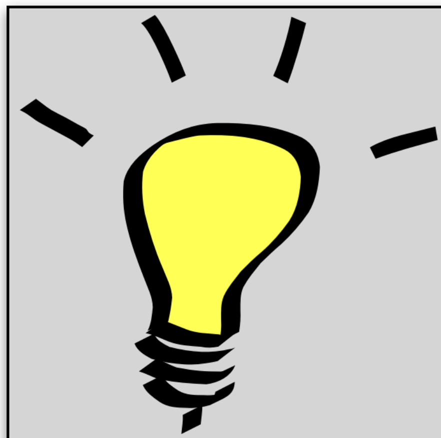


Idea: Making commutativity *explicit* can allow us to **weaken** the dataflow dependency.

Allow the programmer to explicitly express conditional, sequential commutativity.

- Introduce the **commute** keyword.
- Programmer only has to reason **sequentially**.
- Verification tools need only reason **sequentially**.
- Obtain **speedup from parallel** execution.

```
commute (c>0) {
   { x = calc1(a);
     c = c + (x*x);                    }
   { if (c > 0 && y < 0) {
        c = c - 1;
        z = calc2(y);
     } else {
        z = calc3(y);
     }                                 }
```



Idea: Making commutativity *explicit* can allow us to **weaken** the dataflow dependency.

Allow the programmer to explicitly express conditional, sequential commutativity.

- Introduce the **commute** keyword.
- Programmer only has to reason **sequentially**.
- Verification tools need only reason **sequentially**.
- Obtain **speedup from parallel** execution.

```
x = calc1(a);
c = c + (x*x);
```

then

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

||

if(c>0)

else

```
x = calc1(a);
c = c + (x*x);
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

if(c>0)

then

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

||

```
x = calc1(a);
c = c + (x*x);
```

else

```
x = calc1(a);
c = c + (x*x);
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

```
commute (c>0) {
  { x = calc1(a);
    c = c + (x*x);                 }

  { if (c > 0 && y < 0) {
       c = c - 1;
       z = calc2(y);
    } else {
       z = calc3(y);
    }                              }
}
```

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

```
commute (c>0) {
   { x = calc1(a);
     c = c + (x*x);                }
   { if (c > 0 && y < 0) {
        c = c - 1;
        z = calc2(y);
     } else {
        z = calc3(y);
     }                             }
}
```

Semantics?

```
commute (c>0) {
  {
    x = calc1(a);
    c = c + (x*x);
  }
  {
    if (c > 0 && y < 0) {
      c = c - 1;
      z = calc2(y);
    } else {
      z = calc3(y);
    }
  }
}
```

Semantics?

Synchronization?

```
commute (c>0) {
  {
    x = calc1(a);
    c = c + (x*x);
  }
  {
    if (c > 0 && y < 0) {
      c = c - 1;
      z = calc2(y);
    } else {
      z = calc3(y);
    }
  }
}
```

Semantics?

Correct?

Synchronization?

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

```
commute (c>0) {
  {
    x = calc1(a);
    c = c + (x*x);
  }
  {
    if (c > 0 && y < 0) {
      c = c - 1;
      z = calc2(y);
    } else {
      z = calc3(y);
    }
  }
}
```

Semantics?

Correct?

Synchronization?

Speedup?

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

✔ Introducing **`commute`** blocks

2 Semantic Implications & Correctness Criteria — *"Scoped Serializability" and lock synthesis*

3 Demo of the Veracity language — `veracity-lang.org`

4 Speedup

<u>Part B:</u>

5 Symbolic Commutativity Reasoning — *TACAS'18, JAR'20, VMCAI'21*

6 Inferring/verifying **`commute`** blocks — *Translation to logic*

2. Semantic Implications & Correctness Criteria

Want to parallelize a program s.

We can do so soundly when the parallel behavior matches that of its equivalent straight-line code:

$$[\![s]\!]_{nd} = [\![s]\!]_{par}$$

Outcome: Get gains without changing the way we write sequential programs.

Sequential Behavior: non-deterministic

$$\langle \textbf{commute(true)}\{s_1\}\{s_2\}, \sigma\rangle \leadsto_{nd} \langle s_1 ; s_2, \sigma\rangle$$

$$\leadsto_{nd} \langle s_2 ; s_1, \sigma\rangle$$

2. Semantic Implications & Correctness Criteria

Sequential Behavior: non-deterministic

$$\langle \textbf{commute(true)}\{s_1\}\{s_2\}, \sigma\rangle \begin{array}{l} \rightsquigarrow_{nd} \langle s_1; s_2, \sigma\rangle \\ \rightsquigarrow_{nd} \langle s_2; s_1, \sigma\rangle \end{array}$$

Parallel Behavior: Interleaved, as expected:

$$\langle \textbf{commute(true)}\{s_1\}\{s_2\}, \sigma\rangle \rightsquigarrow_{par} \langle ((\langle s_1, \varnothing\rangle, \langle s_2, \varnothing\rangle,), \textbf{skip}, \sigma\rangle$$

$$\frac{\mathfrak{C}_0 \oplus \sigma \rightsquigarrow_{par} \mathfrak{C}_0' \oplus \sigma'}{\langle (\mathfrak{C}_0, \mathfrak{C}_1), s, \sigma\rangle \rightsquigarrow_{par} \langle (\mathfrak{C}_0', \mathfrak{C}_1), s, \sigma'\rangle} \quad \begin{array}{c}\text{Left-Proj} \\ \text{(mut. mut.) R-Proj}\end{array}$$

(Full semantics in the paper)

Sequential Behavior: non-deterministic

$$\langle \textbf{commute(true)}\{s_1\}\{s_2\}, \sigma\rangle \rightsquigarrow_{nd} \langle s_1;s_2, \sigma\rangle$$

$$\langle \textbf{commute(true)}\{s_1\}\{s_2\}, \sigma\rangle \rightsquigarrow_{nd} \langle s_2;s_1, \sigma\rangle$$

Want equivalence (as state fns)

Parallel Behavior: Interleaved, as expected:

$$\langle \textbf{commute(true)}\{s_1\}\{s_2\}, \sigma\rangle \rightsquigarrow_{par} \langle ((\langle s_1, \varnothing\rangle, \langle s_2, \varnothing\rangle,), \textbf{skip}, \sigma\rangle$$

$$\frac{\mathfrak{C}_0 \oplus \sigma \rightsquigarrow_{par} \mathfrak{C}_0' \oplus \sigma'}{\langle(\mathfrak{C}_0, \mathfrak{C}_1), s, \sigma\rangle \rightsquigarrow_{par} \langle(\mathfrak{C}_0', \mathfrak{C}_1), s, \sigma'\rangle} \quad \text{Left-Proj (mut. mut.) R-Proj}$$

(Full semantics in the paper)

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

Serializability?

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

Serializability?

```
y = 0; x = 1;
commute(true) {
  { commute(true)
      f1:{ x = 0; }
      f2:{ x = x*2; }}
 f3:{ if(x>=2) y = 1; }
}
```

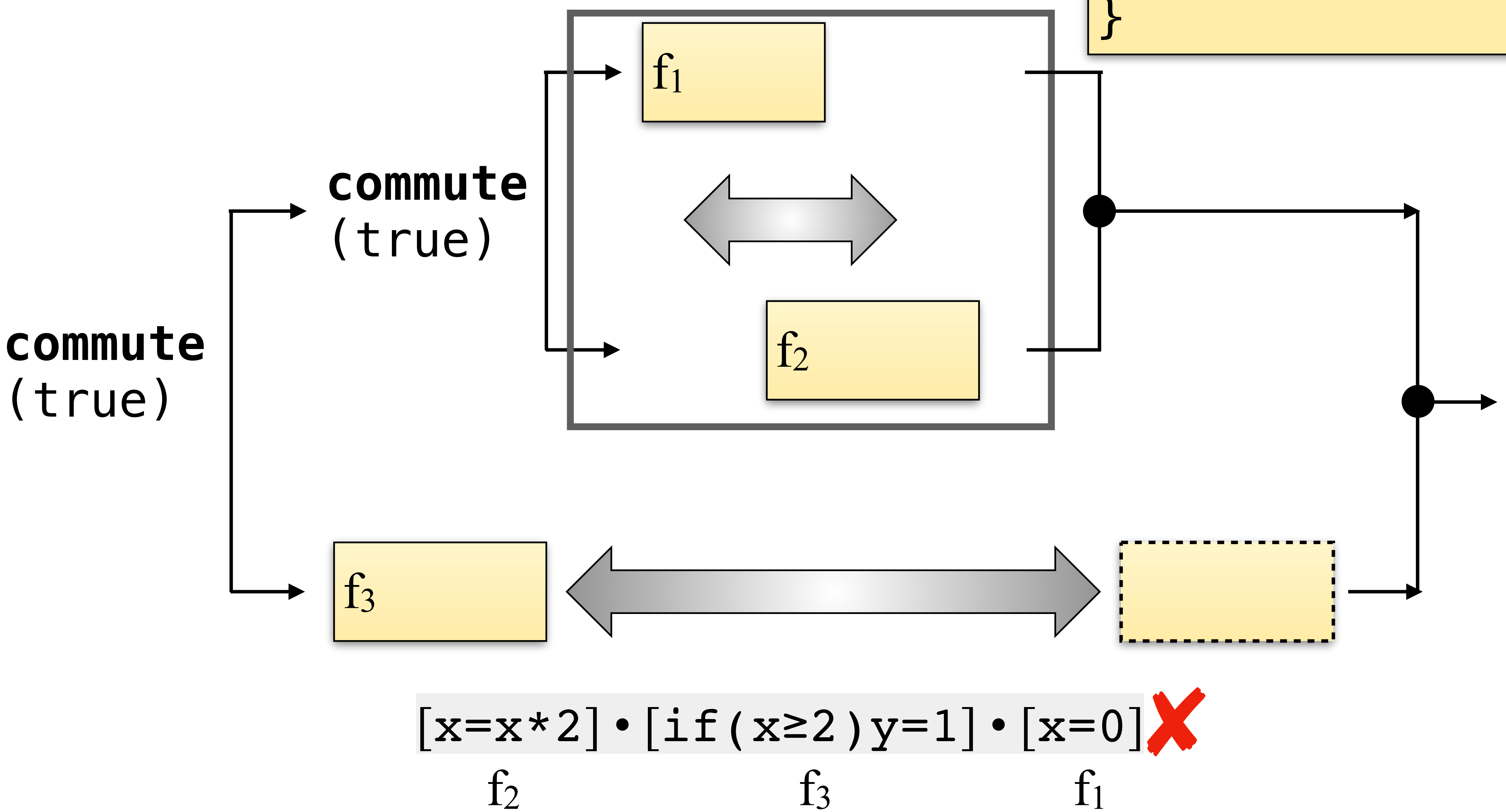
2. Semantic Implications & Correctness Criteria

```
y = 0; x = 1;
commute(true) {
  { commute(true)
      f1:{ x = 0; }
      f2:{ x = x*2; }}
  f3:{ if(x>=2) y = 1; }
}
```

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

Serializability?

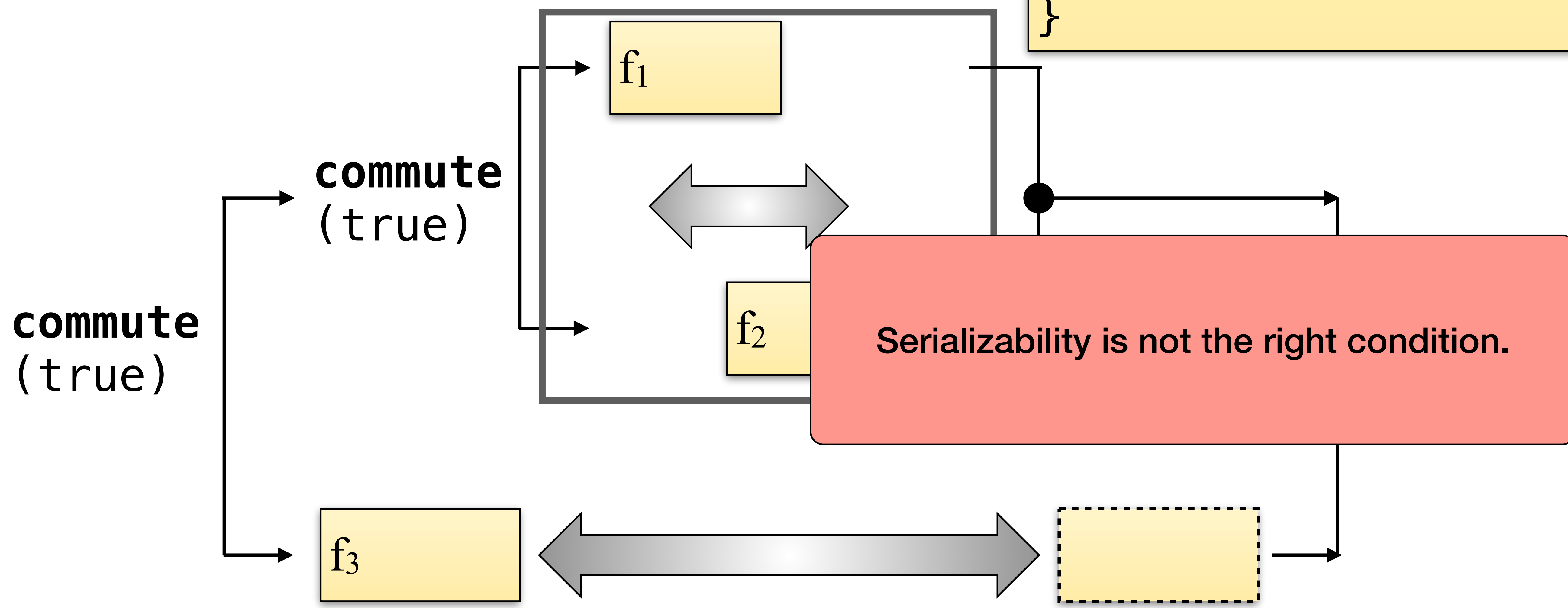


Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

```
y = 0; x = 1;
commute(true) {
  { commute(true)
      f1:{ x = 0; }
      f2:{ x = x*2; }}
  f3:{ if(x>=2) y = 1; }
}
```

How to ensure equivalance.

Serializability?



[x=x*2] • [if(x≥2)y=1] • [x=0] ✗

f_2 f_3 f_1

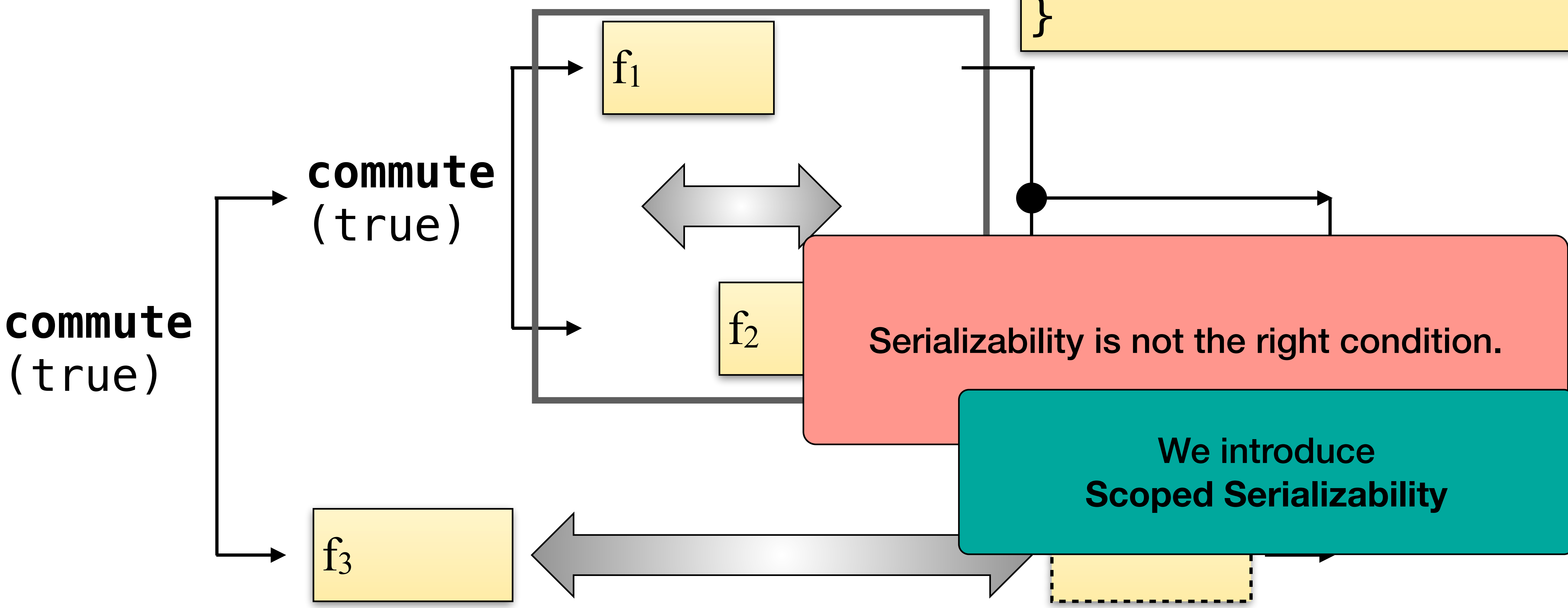
2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

```
y = 0; x = 1;
commute(true) {
  { commute(true)
      f1:{ x = 0; }
      f2:{ x = x*2; }}
  f3:{ if(x>=2) y = 1; }
}
```

How to ensure equivalance.

Serializability?



Serializability is not the right condition.

We introduce
Scoped Serializability

[x=x*2] • [if(x≥2)y=1] • [x=0] ✗

f_2 f_3 f_1

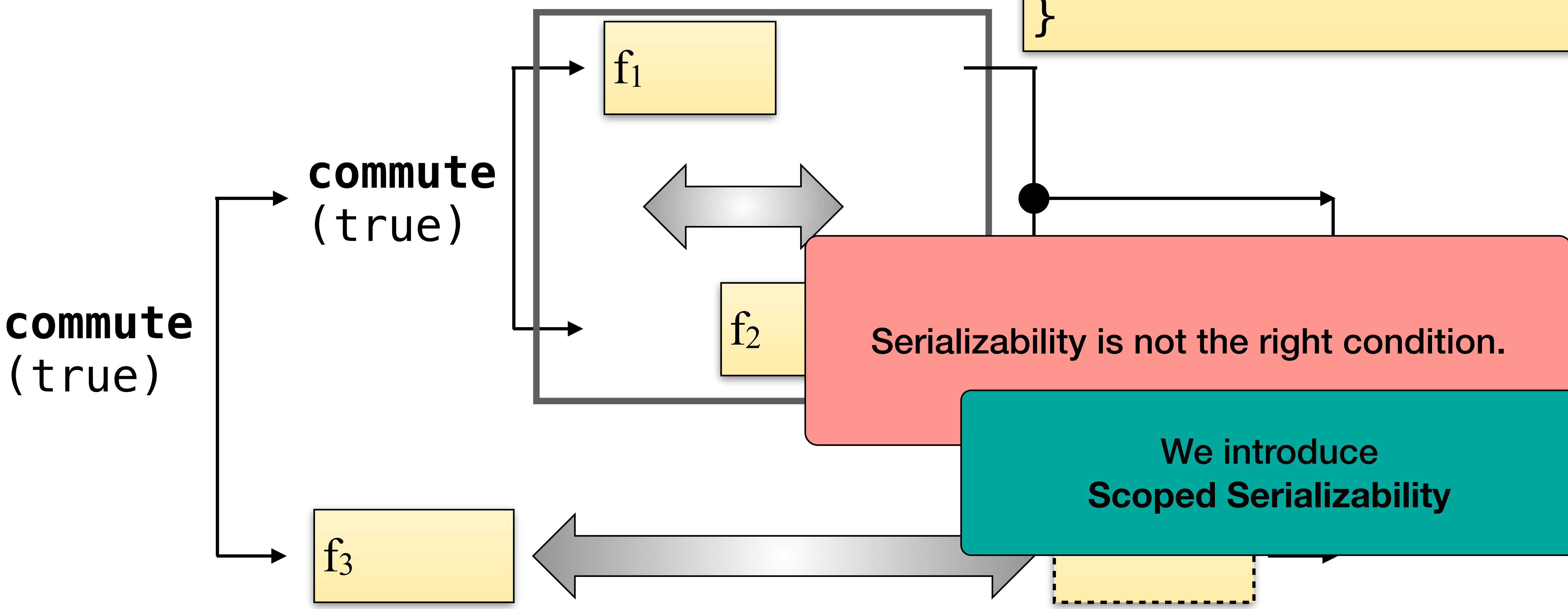
2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

```
y = 0; x = 1;
commute(true) {
  { commute(true)
      f1:{ x = 0; }
      f2:{ x = x*2; }}
  f3:{ if(x>=2) y = 1; }
}
```

How to ensure equivalance.

Serializability?



[x=x*2] • [if(x≥2)y=1] • [x=0] ✗

f_2 f_3 f_1

f_3 can observe intermediate x>=2

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

5.2 Scoped Serializability

We now define our correctness condition. We begin with a single execution:

Definition 5.2 (Scoped serial execution). Execution ε is scoped serial if:

$$\begin{aligned}
&\forall p \in \{L_n, R_n \mid n \in \mathbb{N}\}^* : \\
&((\forall \ell, \ell' \in \varepsilon : \quad \ell.fr \text{ has prefix } p \cdot L_k \wedge \ell'.fr \text{ has prefix } p \cdot R_k \implies \ell \leq_\varepsilon \ell') \\
&\vee(\forall \ell, \ell' \in \varepsilon : \quad \ell.fr \text{ has prefix } p \cdot L_k \wedge \ell'.fr \text{ has prefix } p \cdot R_k \implies \ell' \leq_\varepsilon \ell))
\end{aligned}$$

Above, $\ell.fr$ is the fragment label of ℓ. The key idea here is to identify the *scope* of commute fragments through labels, and then require a serializability condition for the L/R pair of the given scope. Consider, *e.g.*, a single `commute` block, possibly with children. For an execution to be scoped-serial, we require all of the transitions from one of the fragments to execute prior to all the transitions from its co-fragment (the other statement in the `commute`). Next, when there are nested `commute` blocks, the quantification over prefixes requires that we expect this same property to hold locally for all nested `commute` blocks. Without nesting, we recover the standard notion of serial. We now define the following correctness condition, which requires that every execution is equivalent to another scoped serial execution:

Definition 5.3 (Scoped Serializability). For execution ε such that $C \Downarrow_{par} \varepsilon$, ε is scoped serializable,

Intuition: There exists a reordering of every interleaving into a serial order *H* in which *pairs of commute blocks are adjacent* in *H*.

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

```
x = calc1(a);
c = c + (x*x);
```

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

- **Synthesize Locks!**

```
x = calc1(a);
c = c + (x*x);
```

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

- **Synthesize Locks!**

```
x = calc1(a);
lock c = c + (x*x); unlock
```

```
lock if (c > 0 && y < 0) {
        c = c - 1; unlock
        z = calc2(y);
     } else {
        z = calc3(y);
     }
```

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to enforce scoped serializability.**

- **Synthesize Locks!**
- **Use prior works.** e.g. Flanagan & Qadeer 2003, Cherem et al 2008, Vechev et al 2010, Golan-Gueta et al 2015.

```
x = calc1(a);
lock c = c + (x*x); unlock

lock if (c > 0 && y < 0) {
       c = c - 1; unlock
       z = calc2(y);
     } else {
       z = calc3(y);
     }
```

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

- **Synthesize Locks!**
- **Use prior works**. e.g. Flanagan & Qadeer 2003, Cherem et al 2008, Vechev et al 2010, Golan-Gueta et al 2015.
- **Not ideal for commute blocks**.

```
      x = calc1(a);
lock  c = c + (x*x);  unlock
```

```
lock  if (c > 0 && y < 0) {
        c = c - 1;    unlock
        z = calc2(y);
      } else {
        z = calc3(y);
      }
```

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ How to enforce scoped serializability.

- **Synthesize Locks!**
- **Use prior works**. e.g. Flanagan & Qadeer 2003, Cherem et al 2008, Vechev et al 2010, Golan-Gueta et al 2015.
- **Not ideal for commute blocks**.
- **We describe new techniques:**

```
     x = calc1(a);
lock c = c + (x*x); unlock

lock if (c > 0 && y < 0) {
        c = c - 1;  unlock
        z = calc2(y);
     } else {
        z = calc3(y);
     }
```

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

- **Synthesize Locks!**
- **Use prior works**. e.g. Flanagan & Qadeer 2003, Cherem et al 2008, Vechev et al 2010, Golan-Gueta et al 2015.
- **Not ideal for commute blocks**.
- **We describe new techniques:**
  - Achieve shorter windows of locking

```
      x = calc1(a);
lock  c = c + (x*x);  unlock

lock  if (c > 0 && y < 0) {
        c = c - 1;    unlock
        z = calc2(y);
      } else {
        z = calc3(y);
      }
```

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

- **Synthesize Locks!**
- **Use prior works**. e.g. Flanagan & Qadeer 2003, Cherem et al 2008, Vechev et al 2010, Golan-Gueta et al 2015.
- **Not ideal for commute blocks**.
- **We describe new techniques:**
  - Achieve shorter windows of locking
  - Instruction re-ordering

```
     x = calc1(a);
lock c = c + (x*x); unlock
```

```
lock if (c > 0 && y < 0) {
       c = c - 1;       unlock
       z = calc2(y);
     } else {
       z = calc3(y);
     }
```

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

- **Synthesize Locks!**
- **Use prior works**. e.g. Flanagan & Qadeer 2003, Cherem et al 2008, Vechev et al 2010, Golan-Gueta et al 2015.
- **Not ideal for commute blocks**.
- **We describe new techniques:**
  - Achieve shorter windows of locking
  - Instruction re-ordering
  - Details in the paper

```
      x = calc1(a);
lock  c = c + (x*x);  unlock
```

```
lock  if (c > 0 && y < 0) {
        c = c - 1;      unlock
        z = calc2(y);
      } else {
        z = calc3(y);
      }
```

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

Theorem 5.5. If
- every commutativity condition in s is valid,
- and s is scoped-serializable,

Then s is parallelizable, ie, $[\![s]\!]_{nd} = [\![s]\!]_{par}$

2. Semantic Implications & Correctness Criteria

Goal: $[\![s]\!]_{nd} = [\![s]\!]_{par}$

How to ensure equivalance.

☞ **How to <u>enforce</u> scoped serializability.**

Theorem 5.5. If

- every commutativity condition in s is valid,
- and s is scoped-serializable,

Then s is parallelizable, ie, $[\![s]\!]_{nd} = [\![s]\!]_{par}$

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY®

✔ Introducing **`commute`** blocks

✔ Semantic Implications & Correctness Criteria — *"Scoped Serializability" and lock synthesis*

3 Demo of the Veracity language — `veracity-lang.org`

4 Speedup

<u>Part B:</u>

5 Symbolic Commutativity Reasoning — *TACAS'18, JAR'20, VMCAI'21*

6 Inferring/verifying **`commute`** blocks — *Translation to logic*

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Veracity

www.veracity-lang.org

Veracity

`www.veracity-lang.org`

- **New language, implemented in Multicore OCaml**

Veracity

`www.veracity-lang.org`

- **New language, implemented in Multicore OCaml**
- **Interpreter available**. Compiler planned.

Veracity

www.veracity-lang.org

- **New language, implemented in Multicore OCaml**
- **Interpreter available**. Compiler planned.
- **Support for builtin ADTs**. Arrays, Maps/Dictionaries.

Veracity

www.veracity-lang.org

- **New language, implemented in Multicore OCaml**
- **Interpreter available**. Compiler planned.
- **Support for builtin ADTs**. Arrays, Maps/Dictionaries.
- **libcuckoo for parallel maps**. Li *et al,* EuroSys 2014.

Veracity

www.veracity-lang.org

- **New language, implemented in Multicore OCaml**
- **Interpreter available**. Compiler planned.
- **Support for builtin ADTs**. Arrays, Maps/Dictionaries.
- **libcuckoo for parallel maps**. Li *et al,* EuroSys 2014.
- **Underlying SMT solvers**. CVC4, CVC5, Z3.

Veracity

`www.veracity-lang.org`

- **New language, implemented in Multicore OCaml**
- **Interpreter available**. Compiler planned.
- **Support for builtin ADTs**. Arrays, Maps/Dictionaries.
- **libcuckoo for parallel maps**. Li *et al,* EuroSys 2014.
- **Underlying SMT solvers**. CVC4, CVC5, Z3.
- **Examples were written in Veracity**. Execute or inference.

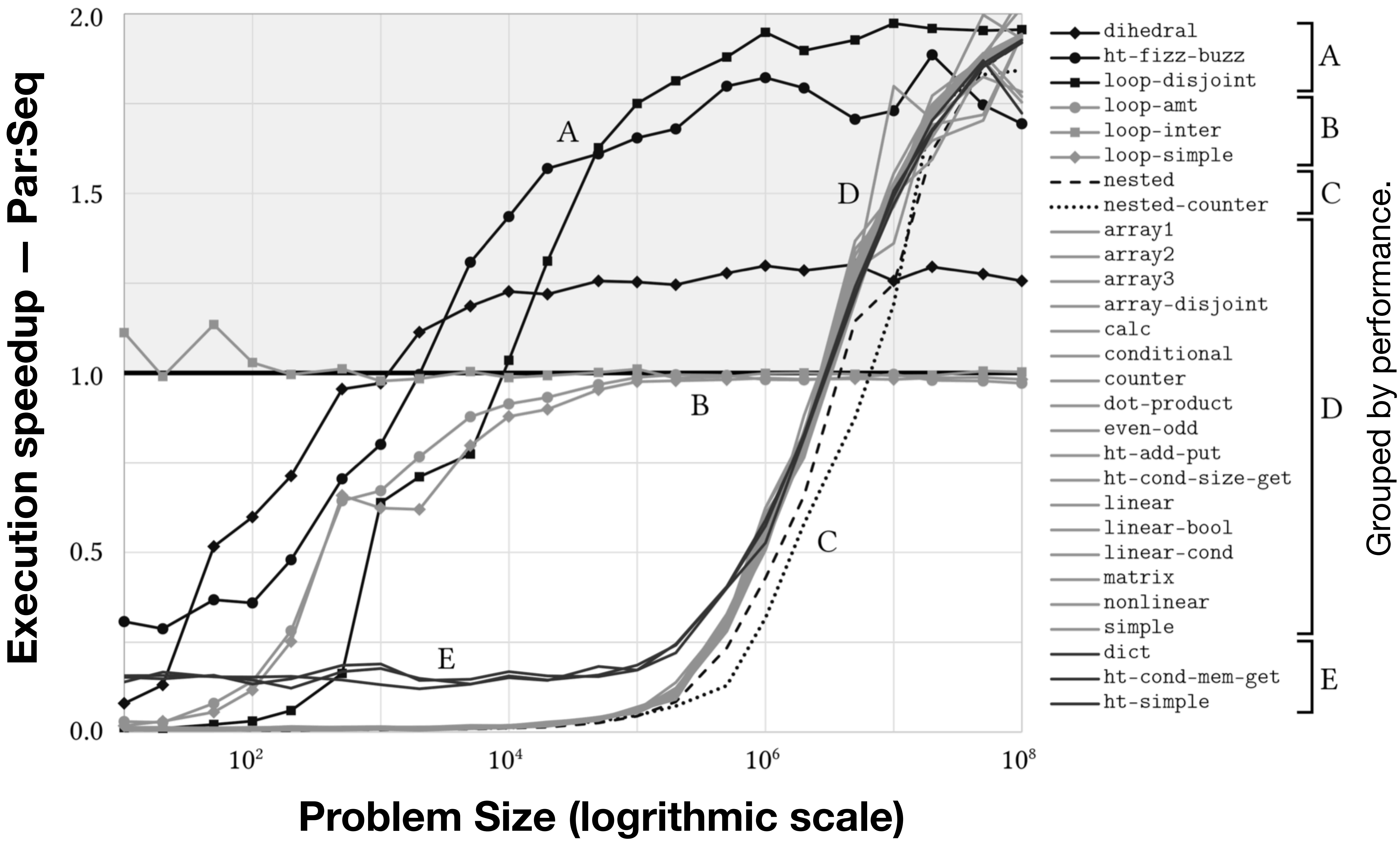
Veracity

`www.veracity-lang.org`

- **New language, implemented in Multicore OCaml**
- **Interpreter available**. Compiler planned.
- **Support for builtin ADTs**. Arrays, Maps/Dictionaries.
- **libcuckoo for parallel maps**. Li *et al,* EuroSys 2014.
- **Underlying SMT solvers**. CVC4, CVC5, Z3.
- **Examples were written in Veracity**. Execute or inference.

```
ejk@arran:veracity/src$ ./vcy.exe interp ../benchmarks/ht-cond-mem-get.vcy 1 2 3 4 5
Return: 0
ejk@arran:veracity/src$
```

Evaluation: Speedup versus problem size

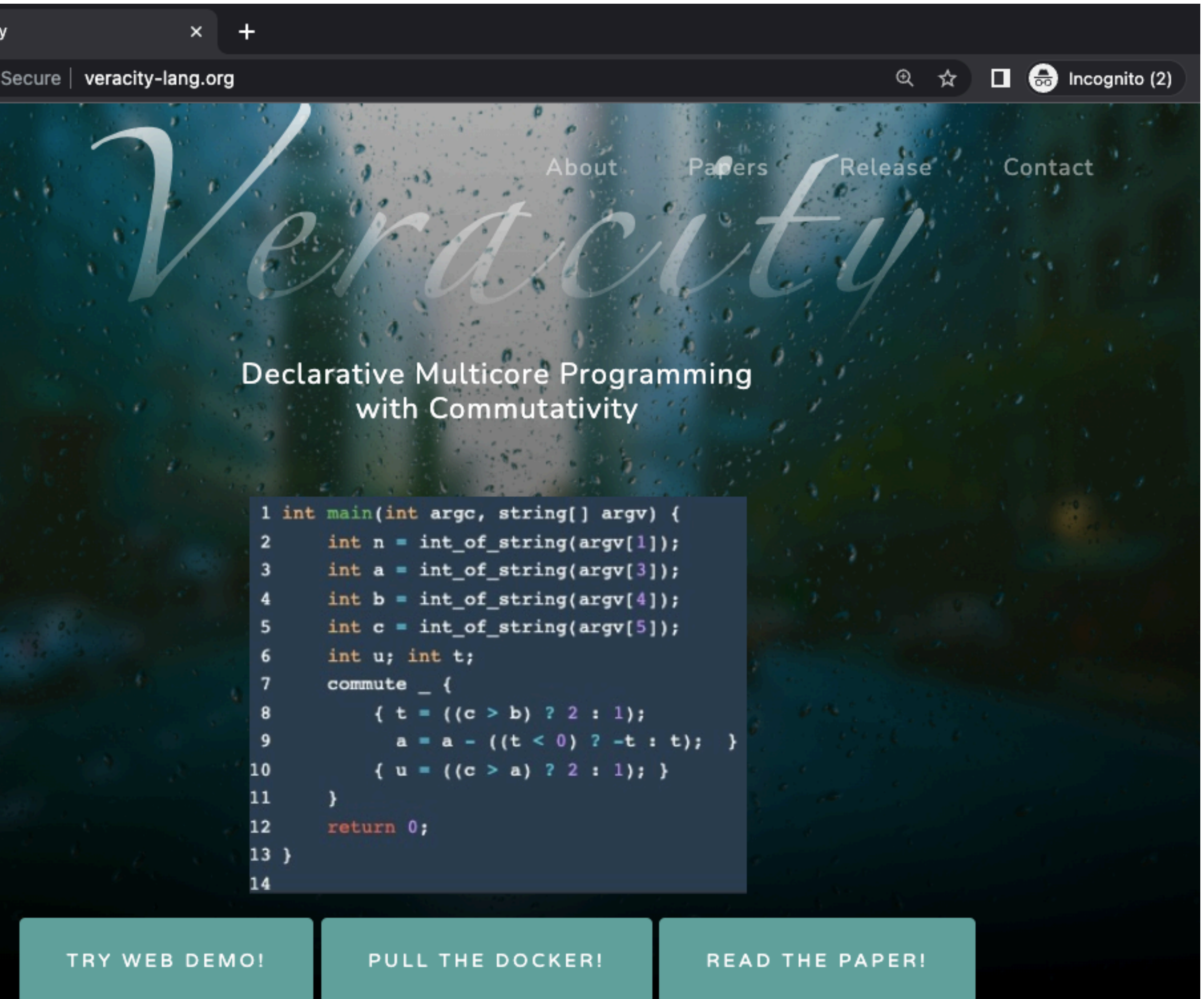


STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Veracity

Programming with Commutativity

www.veracity-lang.org



- Express conditional commutativity.
- Parallel speedup with sequential reasoning.
- New correctness condition.
- Infer or verify commute conditions …

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

- ✔ Introducing `commute` blocks
- ✔ Semantic Implications & Correctness Criteria — *"Scoped Serializability" and lock synthesis*
- ✔ Demo of the Veracity language — `veracity-lang.org`
- ✔ Speedup

Part B:

5 Symbolic Commutativity Reasoning — *TACAS'18, JAR'20, VMCAI'21*

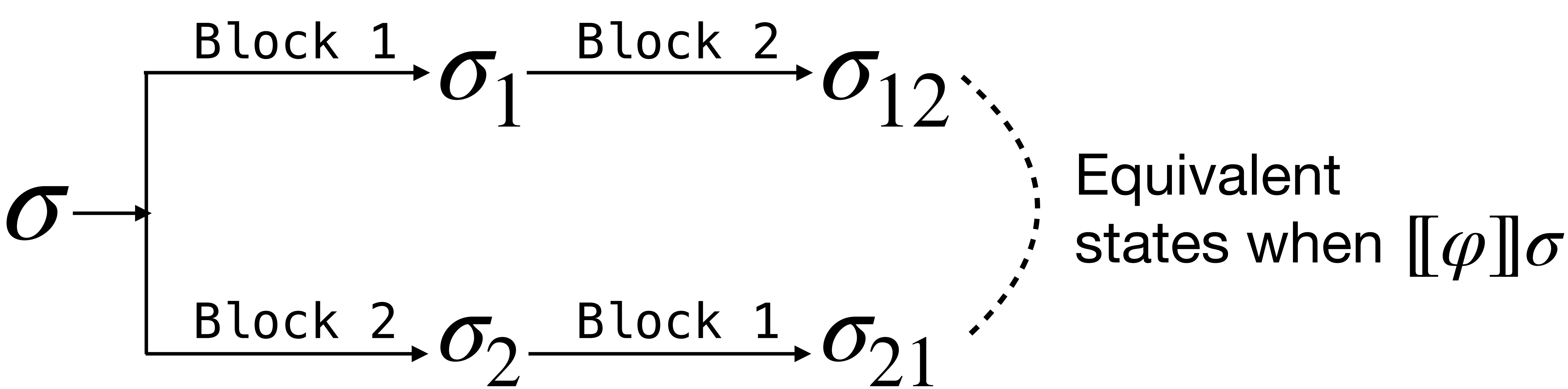
6 Inferring/verifying **`commute`** blocks — *Translation to logic*

```
commute ( φ ) {
  { x = calc1(a);
    c = c + (x*x); }
  { if (c > 0 && y < 0)
         c = c - 1;
       z = calc2(y);
    else
       z = calc3(y); }
}
```

What does it mean for `commute` condition φ to be correct?

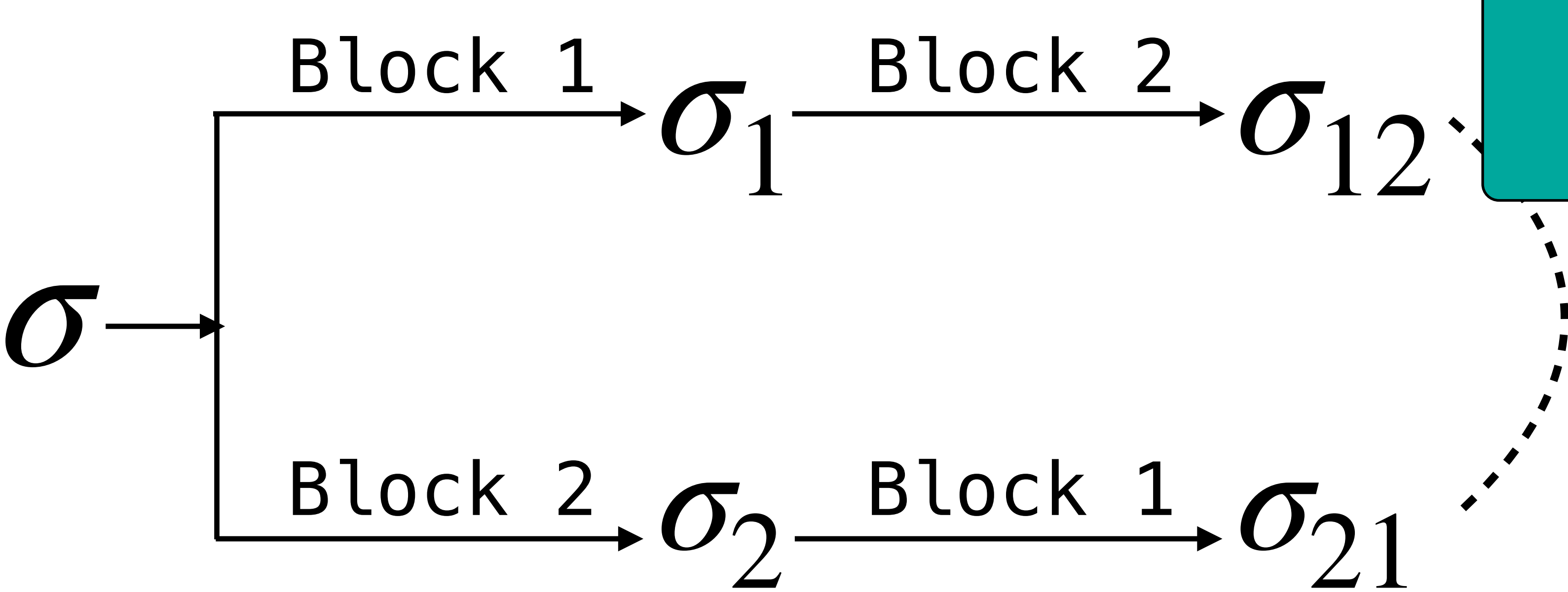
```
commute ( φ ) {
  { x = calc1(a);
    c = c + (x*x); }
  { if (c > 0 && y < 0)
        c = c - 1;
       z = calc2(y);
    else
       z = calc3(y); }
}
```

What does it mean for commute condition φ to be correct?



```
commute ( φ ) {
  { x = calc1(a);
    c = c + (x*x); }
  { if (c > 0 && y < 0)
        c = c - 1;
       z = calc2(y);
    else
       z = calc3(y); }
}
```

What does it mean for `commute` condition φ to be correct?



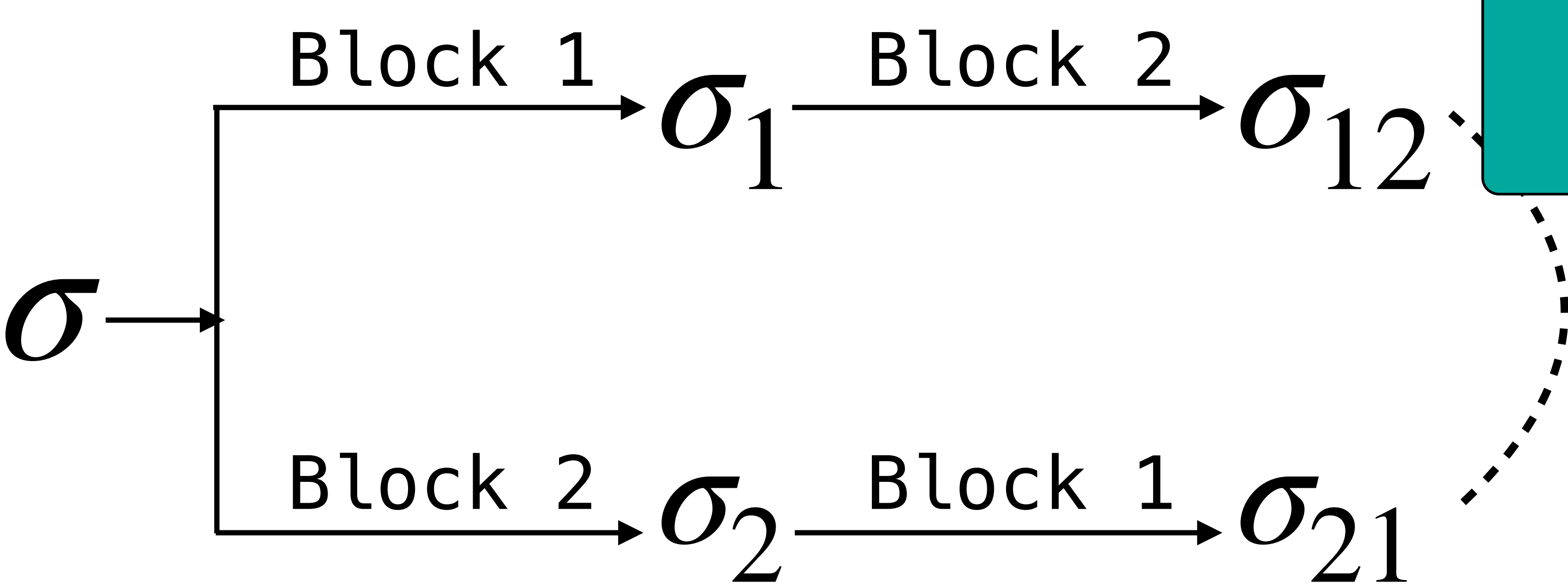
Blocks appear to behave deterministically.

Equivalent states when $[\![\varphi]\!]\sigma$

```
commute ( φ ) {
  { x = calc1(a);
    c = c + (x*x); }
  { if (c > 0 && y < 0)
         c = c - 1;
       z = calc2(y);
    else
       z = calc3(y); }
}
```

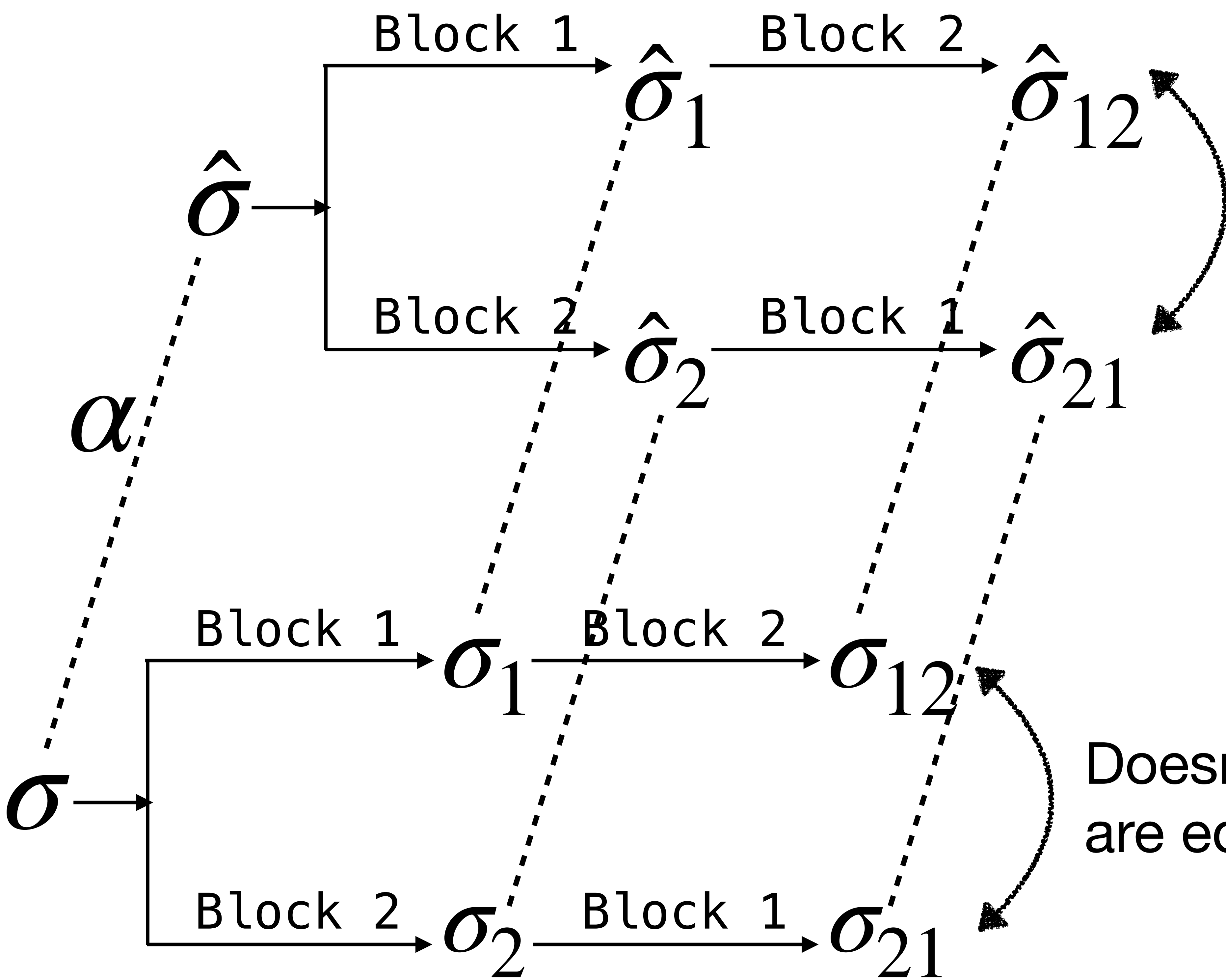
Need symbolic reasoning

What does it mean for commute condition φ to be correct?



Blocks appear to behave deterministically.

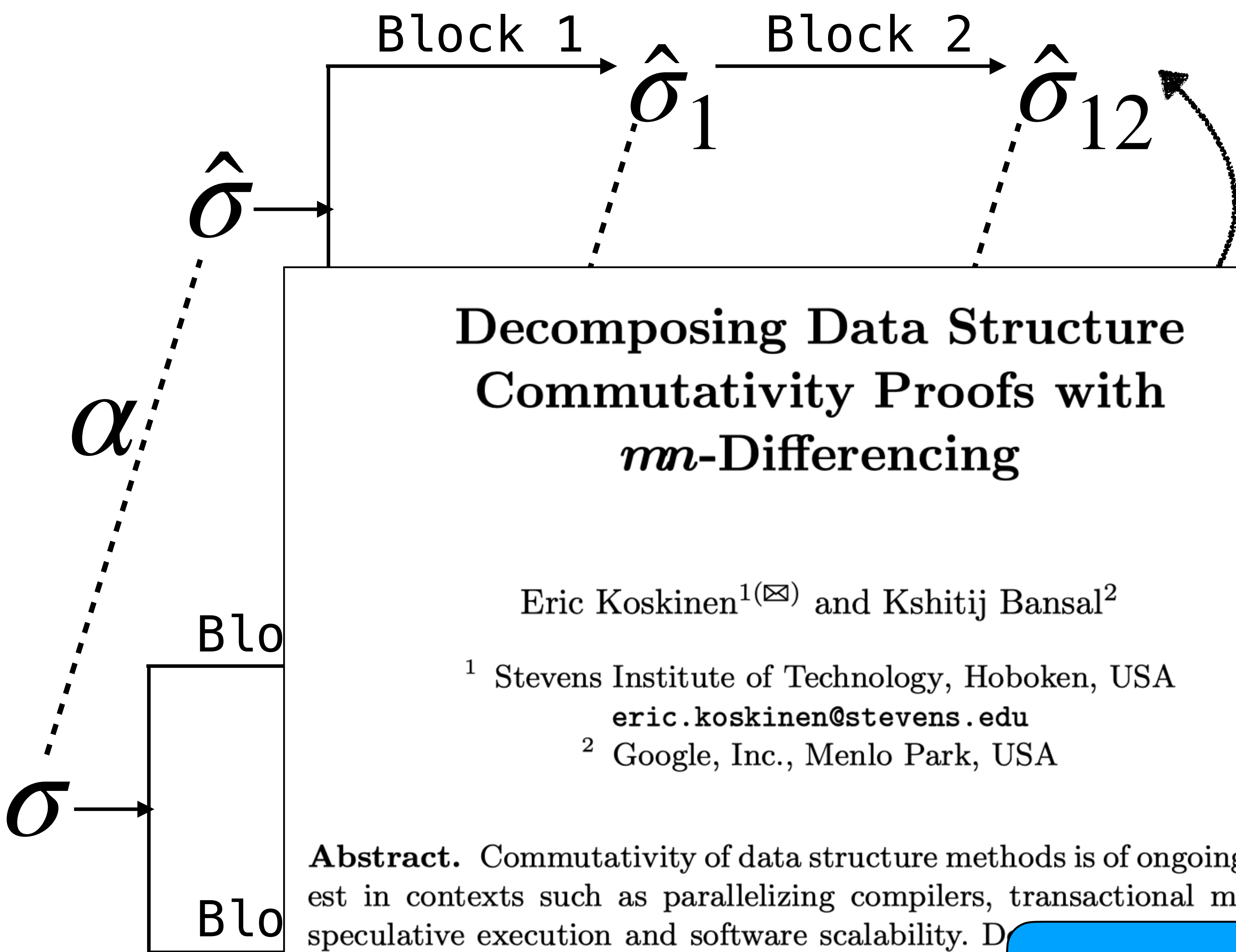
Equivalent states when $[\![\varphi]\!]\sigma$



Just because these are equal

Careful!

Doesn't mean these are equal!



Decomposing Data Structure Commutativity Proofs with *mn*-Differencing

Eric Koskinen[1(✉)] and Kshitij Bansal[2]

[1] Stevens Institute of Technology, Hoboken, USA
eric.koskinen@stevens.edu
[2] Google, Inc., Menlo Park, USA

Abstract. Commutativity of data structure methods is of ongoing interest in contexts such as parallelizing compilers, transactional memory, speculative execution and software scalability. D
lack effective theories and techniques to aid comm
In this paper, we introduce a novel decomposition to improve the task

VMCAI 2021

Just because these are equal

ean these

Careful!

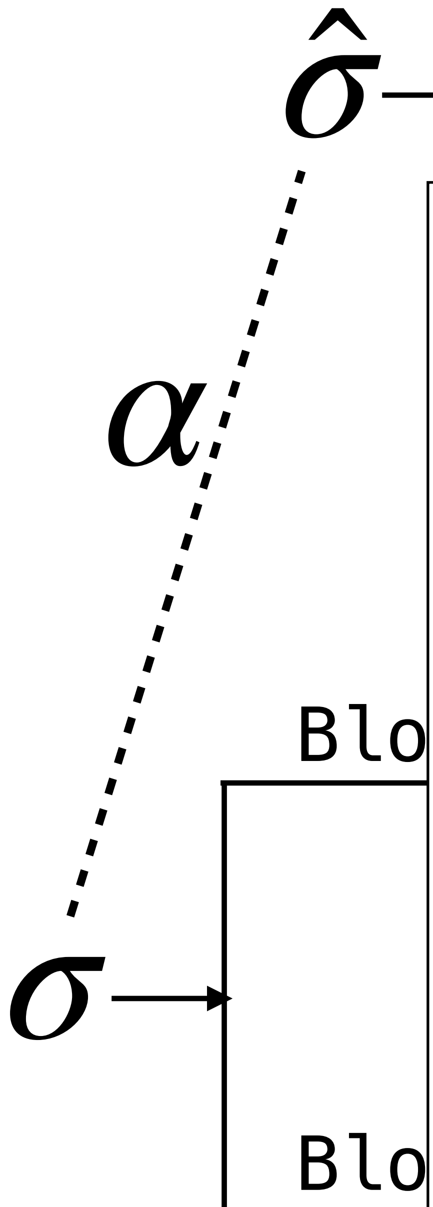
Proof Rule for Decomposing Commutativity Reasoning

$$(i) : \{I_\beta\}[s_m]^1(\bar{x}) \mid [s_m]^2(\bar{y})\{I_\beta\}$$

$$(ii) : \{Rch \wedge \varphi^n_m\} \begin{matrix}[s_m]^1(\bar{x});\\ [s_n]^1(\bar{x})\end{matrix} \mid \begin{matrix}[s_n]^2(\bar{y});\\ [s_m]^2(\bar{y})\end{matrix} \{R_\alpha \wedge C\}$$

$$(iii) : (R_\alpha \wedge C) \implies I_\beta$$

$$\overline{\varphi^n_m \text{ is a commut. cond. for } m(\bar{x}) \bowtie n(\bar{y})}$$



Decomposing Data Structure Commutativity Proofs with *mn*-Differencing

Eric Koskinen[1(✉)] and Kshitij Bansal[2]

[1] Stevens Institute of Technology, Hoboken, USA
eric.koskinen@stevens.edu
[2] Google, Inc., Menlo Park, USA

Abstract. Commutativity of data structure methods is of ongoing interest in contexts such as parallelizing compilers, transactional memory, speculative execution and software scalability. D
lack effective theories and techniques to aid comm
In this paper, we introduce a novel decomposition to improve the task

Careful!

ean these

VMCAI 2021

ADT Impl.

```
insert(){…}
remove(){…}
contains()…
```

Commute Cond.

$$\varphi_m^n$$

Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures

Deokhwan Kim Martin C. Rinard

Massachusetts Institute of Technology
{dkim,rinard}@csail.mit.edu

Abstract

We present a new technique for verifying *commutativity conditions*, which are logical formulas that characterize when operations commute. Because our technique reasons with the abstract state of verified linked data structure implementations, it can verify commuting operations that produce semantically equivalent (but not necessarily identical) data structure states in different execution orders. We have used this technique to verify sound and complete commutativity conditions for all pairs of operations on a collection of linked data structure implementations, including data structures that export a set interface (ListSet and HashSet) as well as data structures that export a map interface (AssociationList, HashTable, and ArrayList). This effort involved the specification and verification of 765 commutativity conditions.

Many speculative parallel systems need to undo the effects of speculatively executed operations. *Inverse operations*, which undo these effects, are often more efficient than alternate approaches (such as saving and restoring data structure state). We present a new technique for verifying such inverse operations. We have specified and verified, for all of our linked data structure implementations, an inverse operation for every operation that changes the data structure state.

Together, the commutativity conditions and inverse operations provide a key resource that language designers, developers of program analysis systems, and implementors of software systems can draw on to build languages, program analyses, and systems with strong correctness guarantees.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.4 [*Software Engineering*]:

- **Deterministic Parallel Languages:** Including support for commuting operations in deterministic parallel languages increases the expressive power of the language while preserving guaranteed deterministic parallel execution [5, 42].
- **Transaction Monitors:** If a transaction monitor can detect that operations within parallel transactions commute, it can use efficient locking algorithms that allow commuting operations from different transactions to interleave [17, 49]. Because such locking algorithms place fewer constraints on the execution order, they increase the amount of exploitable parallelism.
- **Irregular Parallel Computations:** Exploiting commuting operations has been shown to be critical for obtaining good parallel performance in irregular parallel computations that manipulate linked data structures [28–30]. The reason is essentially the same as for efficient transaction monitors — it enables the use of efficient synchronization algorithms for atomic transactions that execute multiple (potentially commuting) operations on shared objects. For similar reasons, exploiting commuting operations has also been shown to be essential for obtaining good parallel performance for the SPEC benchmarks [7].

Despite the importance of commuting operations, there has been relatively little research in automatically analyzing or verifying the conditions under which operations commute. Indeed, the deterministic parallel language, transaction monitor, and irregular parallel computation systems cited above all rely on the developer to identify commuting operations, with no way to determine whether the operations do, in fact, commute or not. A mistake in identifying commuting operations invalidates both the principles upon which the systems operate and the correctness guarantees that they claim

ADT Impl.

```
insert(){…}
remove(){…}
contains()…
```

Commute Cond.

$$\varphi_m^n$$

Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures

Deokhwan Kim Martin C. Rinard
Massachusetts Institute of Technology
{dkim,rinard}@csail.mit.edu

…rallel Languages: Including support for …tions in deterministic parallel languages in… …sive power of the language while preserving …inistic parallel execution [5, 42].

…itors: If a transaction monitor can detect …thin parallel transactions commute, it can use …lgorithms that allow commuting operations …sactions to interleave [17, 49]. Because such …s place fewer constraints on the execution …e the amount of exploitable parallelism.

…l Computations: Exploiting commuting op… …shown to be critical for obtaining good paral… …i irregular parallel computations that manip… …structures [28–30]. The reason is essentially …ficient transaction monitors — it enables the …nchronization algorithms for atomic transac… …multiple (potentially commuting) operations … For similar reasons, exploiting commuting …o been shown to be essential for obtaining …ormance for the SPEC benchmarks [7].

…ance of commuting operations, there has been …h in automatically analyzing or verifying the …h operations commute. Indeed, the determin…, transaction monitor, and irregular parallel …cited above all rely on the developer to iden… …tions, with no way to determine whether the …, commute or not. A mistake in identifying …s invalidates both the principles upon which …nd the correctness guarantees that they claim

Jahob Verification System

Jahob is a verification system for programs written in a subset of Java. Using Ja… methods satisfy their contracts in all possible executions, as well as that they p… design constraints.

Jahob is now on github: https://github.com/epfl-lara/jahob

Note

- Information below may be outdated
- Java may be outdated. Consider Scala, http://www.scala-lang.org/
- To verify Scala, consider tools such as http://leon.epfl.ch and its successor…

Some of the data structures verified in Jahob

You may wish to compare

- ArrayList class JavaDoc to

ADT Impl.

```
insert(){…}
remove(){…}
contains()…
```

↔

ADT Spec.

$P_{\text{ins}}, Q_{\text{ins}}$

$P_{\text{rem}}, Q_{\text{rem}}$

$P_{\text{con}}, Q_{\text{con}}$

↔

Commute Cond.

φ_m^n

Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures

Deokhwan Kim Martin C. Rinard
Massachusetts Institute of Technology
{dkim,rinard}@csail.mit.edu

…rallel Languages: Including support for …tions in deterministic parallel languages in…sive power of the language while preserving …inistic parallel execution [5, 42].

…itors: If a transaction monitor can detect …thin parallel transactions commute, it can use …lgorithms that allow commuting operations …sactions to interleave [17, 49]. Because such …s place fewer constraints on the execution …e the amount of exploitable parallelism.

…l Computations: Exploiting commuting op… …shown to be critical for obtaining good paral… …irregular parallel computations that manip… …structures [28–30]. The reason is essentially …ficient transaction monitors — it enables the …nchronization algorithms for atomic transac… …multiple (potentially commuting) operations …. For similar reasons, exploiting commuting …o been shown to be essential for obtaining …rmance for the SPEC benchmarks [7].

…ance of commuting operations, there has been …h in automatically analyzing or verifying the …h operations commute. Indeed, the determin… …, transaction monitor, and irregular parallel …cited above all rely on the developer to iden… …tions, with no way to determine whether the …, commute or not. A mistake in identifying …s invalidates both the principles upon which …nd the correctness guarantees that they claim

Jahob Verification System

Jahob is a verification system for programs written in a subset of Java. Using J… methods satisfy their contracts in all possible executions, as well as that they p… design constraints.

Jahob is now on github: https://github.com/epfl-lara/jahob

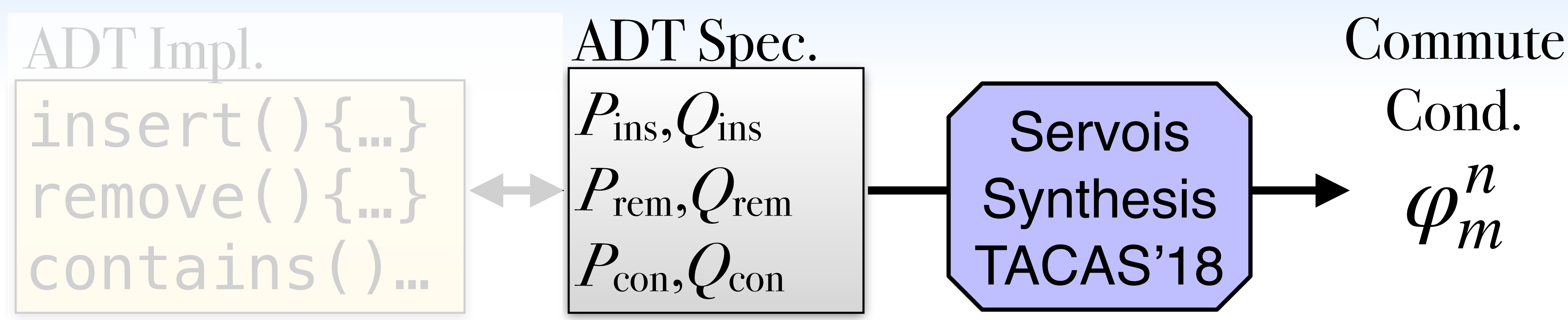
Note

- Information below may be outdated
- Java may be outdated. Consider Scala, http://www.scala-lang.org/
- To verify Scala, consider tools such as http://leon.epfl.ch and its successo…

Some of the data structures verified in Jahob

You may wish to compare

- ArrayList class JavaDoc to

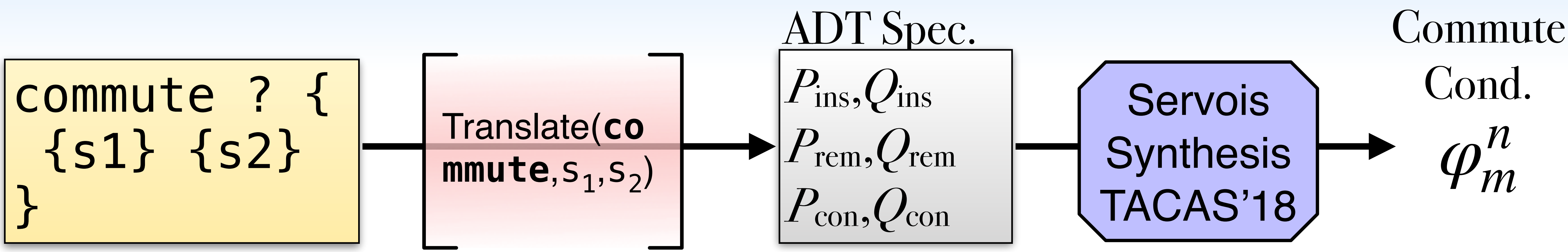


Automatic Generation of Precise and Useful Commutativity Conditions

Kshitij Bansal[1*], Eric Koskinen[2†], and Omer Tripp[1‡]

	$m(\bar{x})$	$n(\bar{y})$	Simple Qs (time)	Poke Qs (time)	φ_n^m (Poke)
Counter	decrement	⋈ decrement	3 (0.1)	3 (0.1)	true
	increment	▷ decrement	10 (0.3)	34 (0.9)	$\neg(0 = c)$
	decrement	▷ increment	3 (0.1)	3 (0.1)	true
	decrement	⋈ reset	2 (0.1)	2 (0.1)	false
	decrement	⋈ zero	6 (0.1)	26 (0.6)	$\neg(1 = c)$
	increment	⋈ increment	3 (0.1)	3 (0.1)	true
	increment	⋈ reset	2 (0.0)	2 (0.1)	false
	increment	⋈ zero	10 (0.3)	34 (0.8)	$\neg(0 = c)$
	reset	⋈ reset	3 (0.1)	3 (0.1)	true
	reset	⋈ zero	9 (0.2)	30 (0.6)	$0 = c$
	zero	⋈ zero	3 (0.1)	3 (0.1)	true
Acum.	increase	⋈ increase	3 (0.1)	3 (0.1)	true
	increase	⋈ read	13 (0.3)	28 (0.6)	$c + x_1 = c$
	read	⋈ read	3 (0.0)	3 (0.0)	true
Set	add	⋈ add	10 (0.4)	140 (4.4)	$(y_1 = x_1 \wedge y_1 \in S) \vee \neg(y_1 = x_1)$
	add	⋈ contains	10 (0.4)	122 (3.6)	$x_1 \in S \vee (\neg(x_1 \in S) \wedge \neg(y_1 = x_1))$
	add	⋈ getsize	6 (0.2)	31 (0.9)	$x_1 \in S$
	add	⋈ remove	6 (0.2)	66 (2.2)	$\neg(y_1 = x_1)$
	contains	⋈ contains	3 (0.1)	3 (0.1)	true
	contains	⋈ getsize	3 (0.1)	3 (0.1)	true
	contains	⋈ remove	17 (0.5)	160 (4.8)	$S \setminus \{x_1\} = \{y_1\} \vee (\ldots \wedge y_1 \in \{x_1\}) \vee \ldots$

TACAS 2018



Veracity: Declarative Multicore Programming with Commutativity

ADAM CHEN, Stevens Institute of Technology, USA
PARISA FATHOLOLUMI, Stevens Institute of Technology, USA
ERIC KOSKINEN, Stevens Institute of Technology, USA
JARED PINCUS, Stevens Institute of Technology, USA

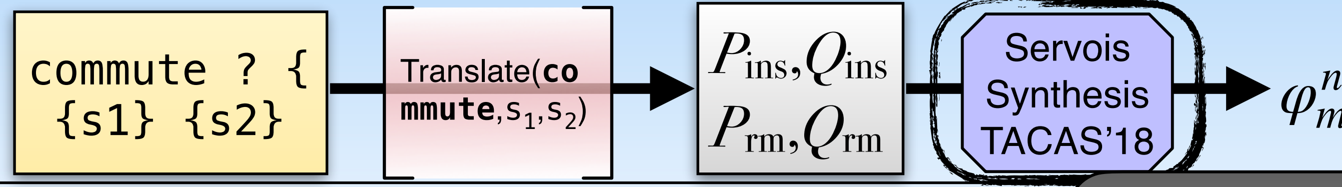
There is an ongoing effort to provide programming abstractions that ease the burden of exploiting multicore hardware. Many programming abstractions (*e.g.*, concurrent objects, transactional memory, etc.) simplify matters, but still involve intricate engineering. We argue that some difficulty of multicore programming can be meliorated through a declarative programming style in which programmers directly express the independence of fragments of sequential programs.

In our proposed paradigm, programmers write programs in a familiar, sequential manner, with the added ability to explicitly express the conditions under which code fragments sequentially commute. Putting such commutativity conditions into source code offers a new entry point for a compiler to exploit the known connection between commutativity and parallelism. We give a semantics for the programmer's sequential perspective and, under a correctness condition, find that a compiler-transformed parallel execution is equivalent to the sequential semantics. Serializability/linearizability are not the right fit for this condition, so we introduce scoped serializability and show how it can be enforced with lock synthesis techniques.

We next describe a technique for automatically verifying and synthesizing commute conditions via a new reduction from our commute blocks to logical specifications, upon which symbolic commutativity reasoning can be performed. We implemented our work in a new language called Veracity, implemented in Multicore OCaml. We show that commutativity conditions can be automatically generated across a variety of new benchmark programs, confirm the expectation that concurrency speedups can be seen as the computation increases, and apply our work to a small in-memory filesystem and an adaptation of a crowdfund blockchain smart contract.

1 INTRODUCTION

Writing concurrent programs is difficult. Researchers and practitioners, seeking to make life easier,



TACAS'18/JAR'20

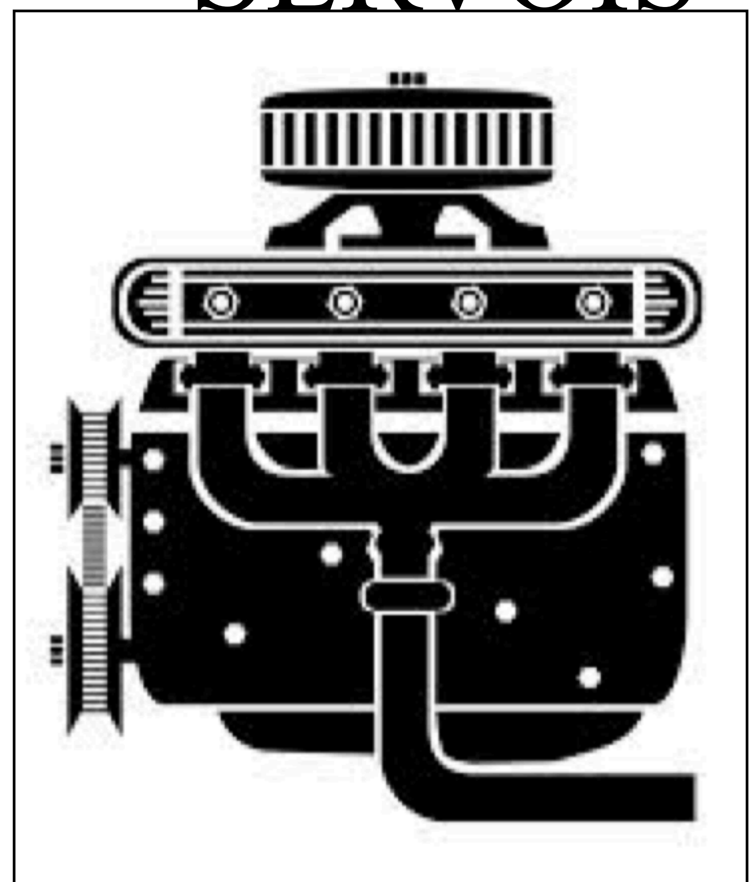
STEVENS INSTITUTE *of* TECHNOLOGY
THE INNOVATION UNIVERSITY®

`commute ? { {s1} {s2}` → Translate(**commute**,s_1,s_2) → $P_{\mathrm{ins}}, Q_{\mathrm{ins}}$ / $P_{\mathrm{rm}}, Q_{\mathrm{rm}}$ → Servois Synthesis TACAS'18 → φ_m^n

Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{llll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

SERVOIS



Automatic Generation of Precise and Useful Commutativity Conditions

Kshitij Bansal[1*], Eric Koskinen[2†], and Omer Tripp[1‡]

[1] Google, Inc.
[2] Stevens Institute of Technology

Abstract. Reasoning about commutativity between data-structure operations is an important problem with applications including parallelizing compilers, optimistic parallelization and, more recently, Ethereum smart contracts. There have been research results on automatic generation of commutativity conditions, yet we are unaware of any fully automated technique to generate conditions that are both sound and effective.
We have designed such a technique, driven by an algorithm that iteratively refines a conservative approximation of the commutativity (and non-commutativity) condition for a pair of methods into an increasingly precise version. The algorithm terminates if/when the entire state space has been considered, and can be aborted at any time to obtain a partial yet sound commutativity condition. We have generalized our work to left-/right-movers [27] and proved relative completeness. We describe aspects of our technique that lead to *useful* commutativity conditions, including how predicates are selected during refinement and heuristics that impact the output shape of the condition.
We have implemented our technique in a prototype open-source tool SERVOIS. Our algorithm produces quantifier-free queries that are dispatched to a back-end SMT solver. We evaluate SERVOIS through two case studies: (i) We synthesize commutativity conditions for a range of data structures including Set, HashTable
We consider an Ethereum sma
that SERVOIS can detect seriou
guide developers to construct r

TACAS 2018, JAR 2020

$$(\varphi_m^n, \overline{\varphi_m^n})$$

Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{lll lll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$ for methods $m(\bar{x})/r_m,\ n(\bar{y})/r_n$

valid $\left\{ \varphi^n_m(\sigma, \bar{x}, \bar{y}) \implies m(\bar{x})/r_m \bowtie n(\bar{y})/r_n \right\}$

...

Logical ADT Specification

$$O = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \to (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$ for methods $m(\bar{x})/r_m,\ n(\bar{y})/r_n$

$$\textbf{valid} \left\{ \varphi^n_m(\sigma, \bar{x}, \bar{y}) \implies m(\bar{x})/r_m \bowtie n(\bar{y})/r_n \right\}$$

$$\forall \sigma_0, \sigma_1, \sigma_2, x, y, r_m, r_n.$$

...

Logical ADT Specification

$$O = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$ for methods $m(\bar{x})/r_m,\ n(\bar{y})/r_n$

valid $\left\{ \varphi_m^n(\sigma, \bar{x}, \bar{y}) \implies m(\bar{x})/r_m \bowtie n(\bar{y})/r_n \right\}$

$$\forall \sigma_0, \sigma_1, \sigma_2, x, y, r_m, r_n.$$
$$\sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2 \implies$$

...

Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \quad \text{methods} : \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \quad \text{spec} : \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$ for methods $m(\bar{x})/r_m,\ n(\bar{y})/r_n$

$$\textbf{valid} \left\{ \varphi_m^n(\sigma, \bar{x}, \bar{y}) \implies m(\bar{x})/r_m \bowtie n(\bar{y})/r_n \right\}$$

$$\forall \sigma_0, \sigma_1, \sigma_2, x, y, r_m, r_n.$$
$$\sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2 \implies$$
$$(\exists \sigma_3 . \sigma_0 \xrightarrow{n(y)/r_n} \sigma_3 \xrightarrow{m(x)/r_m} \sigma_2)) \wedge \ldots$$

Logical ADT Specification

$$O = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \to (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$ for methods $m(\bar{x})/r_m$, $n(\bar{y})/r_n$

$$\textbf{valid} \left\{ \varphi_m^n(\sigma, \bar{x}, \bar{y}) \implies m(\bar{x})/r_m \bowtie n(\bar{y})/r_n \right\}$$

$$\forall \sigma_0, \sigma_1, \sigma_2, x, y, r_m, r_n . \quad \sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2 \implies (\exists \sigma_3 . \sigma_0 \xrightarrow{n(y)/r_n} \sigma_3 \xrightarrow{m(x)/r_m} \sigma_2)) \wedge \ldots$$

Quantifier alternation

Logical ADT Specification

$$O = \left\{ \begin{array}{llll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \to (P, Q) \end{array} \right\}$$

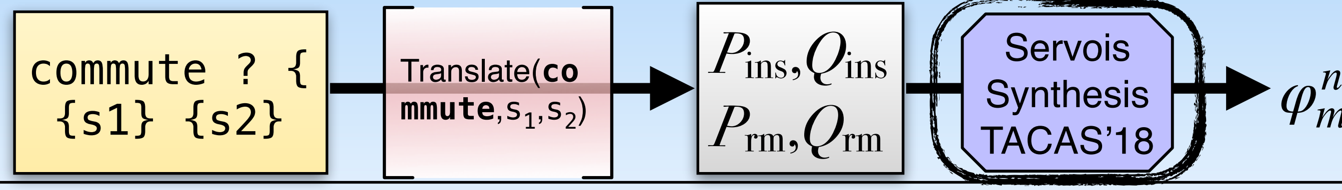
1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$ for methods $m(\bar{x})/r_m,\ n(\bar{y})/r_n$

$$\textbf{valid} \left\{ \varphi_m^n(\sigma, \bar{x}, \bar{y}) \implies m(\bar{x})/r_m \bowtie n(\bar{y})/r_n \right\}$$

$$\forall \sigma_0, \sigma_1, \sigma_2, x, y, r_m, r_n .$$
$$\sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2 \implies$$
$$(\exists \sigma_3 . \sigma_0 \xrightarrow{n(y)/r_n} \sigma_3 \xrightarrow{m(x)/r_m} \sigma_2)) \wedge \ldots$$

Quantifier alternation

Avoid introducing quantifiers in encoding of commutativity



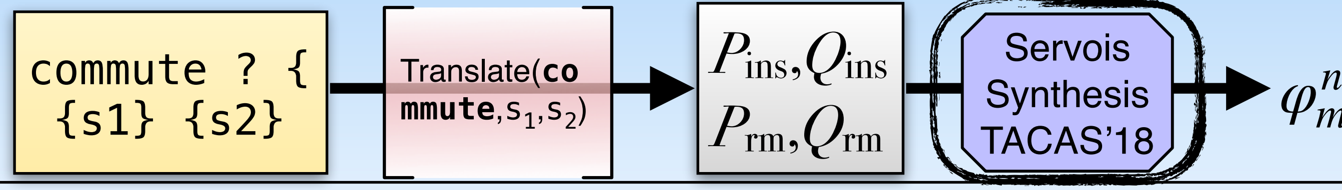
Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{lllllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \longrightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition

Use a form of abstraction-refinement.
Start with candidate commutativity condition H

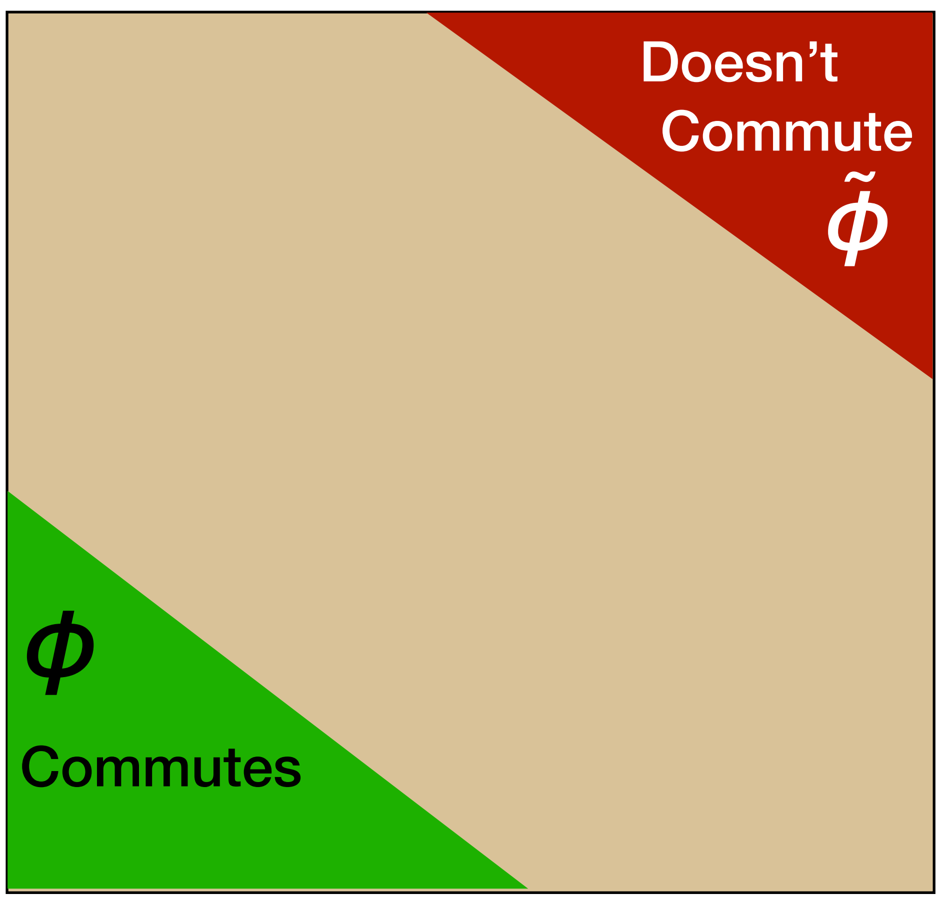
$$\textbf{valid} \left\{ H \implies m(\bar{x})/r_m \bowtie n(\bar{y})/r_n \right\}$$



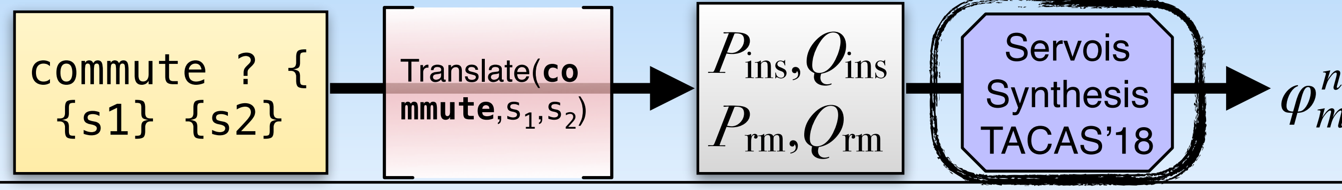
Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{llllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



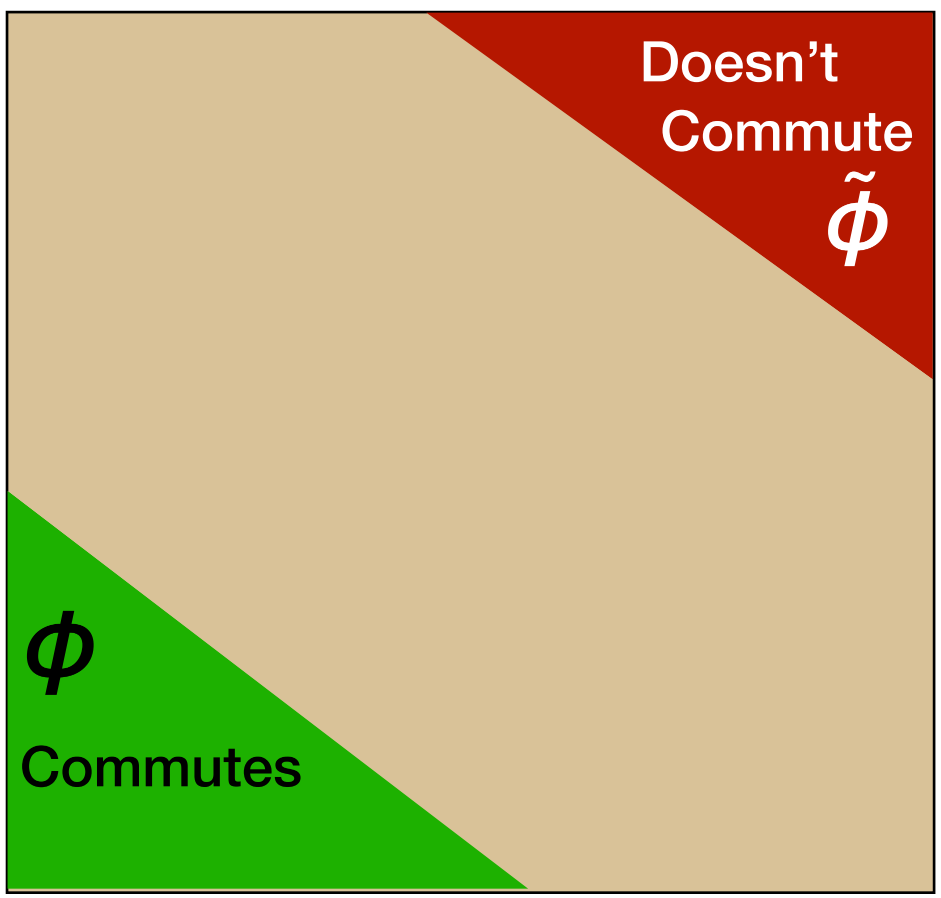
Refine(H)



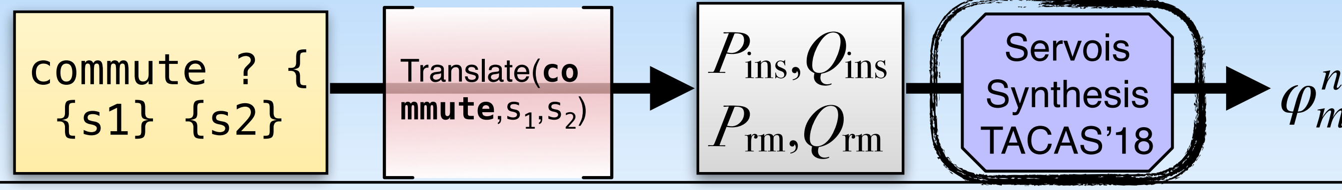
Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{llll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



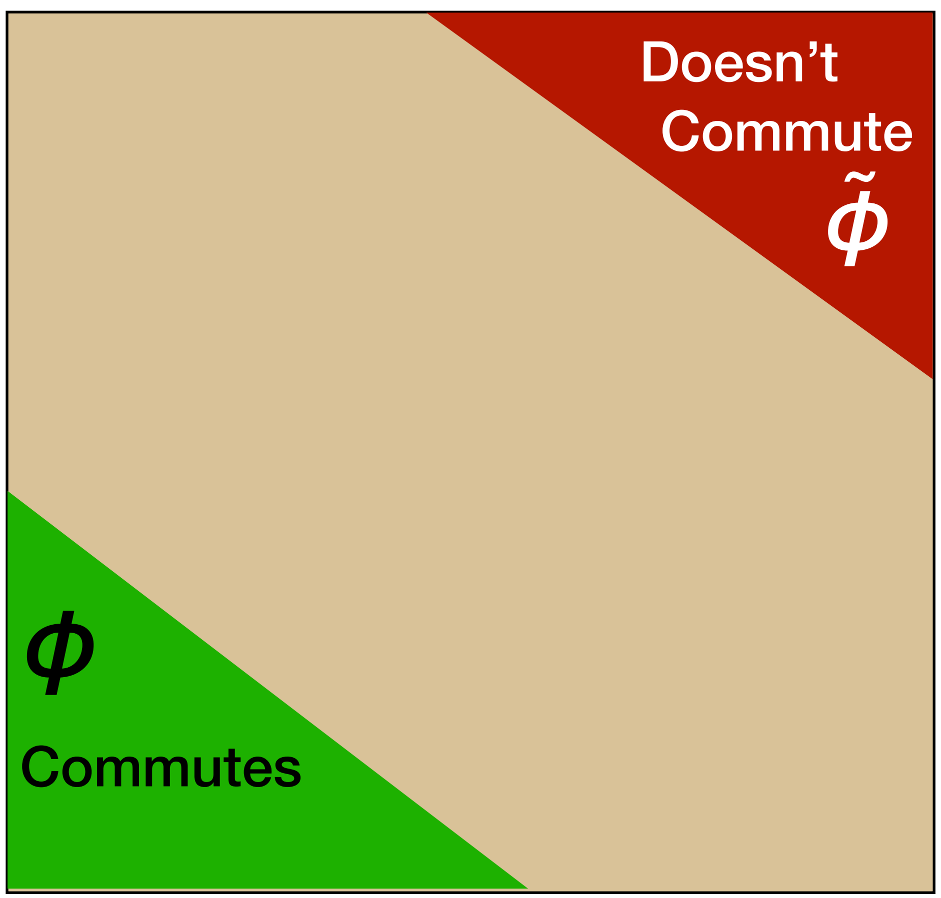
Refine(H)



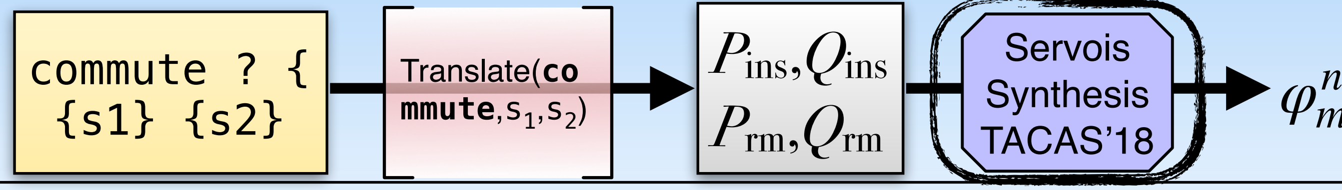
Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{llll} \text{state} : (Var, Type)\text{list}; & \text{methods} : \text{Meth list}; \\ \text{eq} \subseteq \text{state} \times \text{state}; & \text{spec} : \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



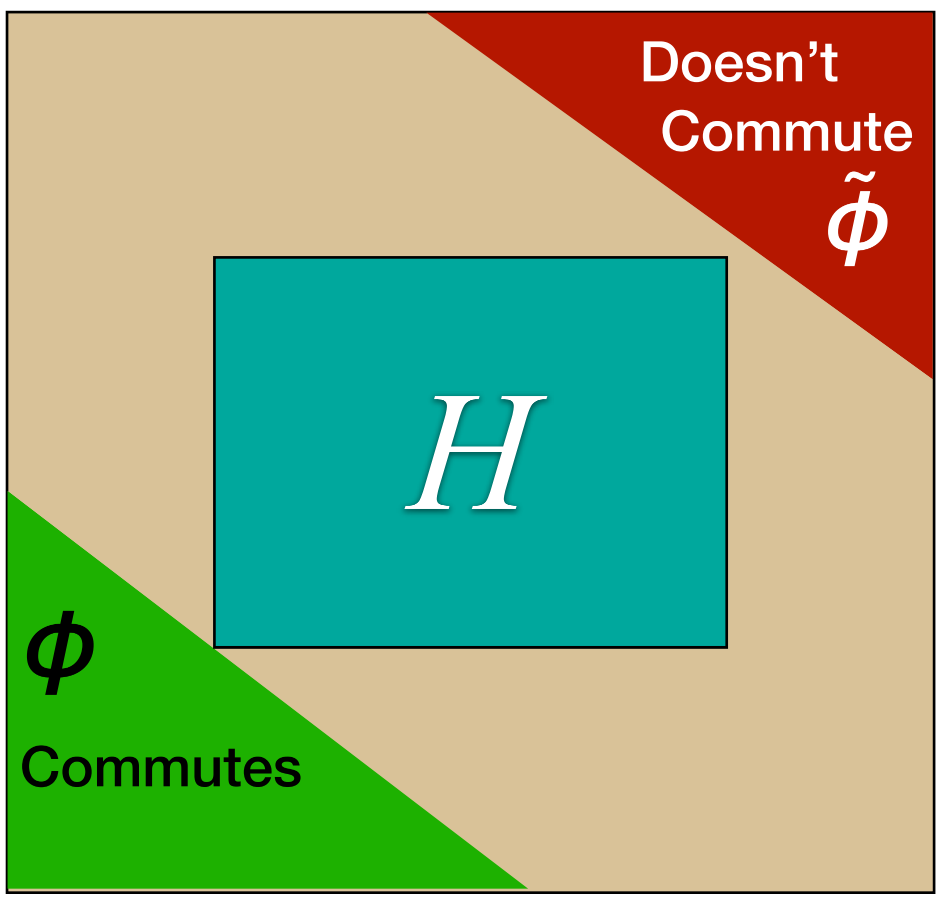
Refine(H)



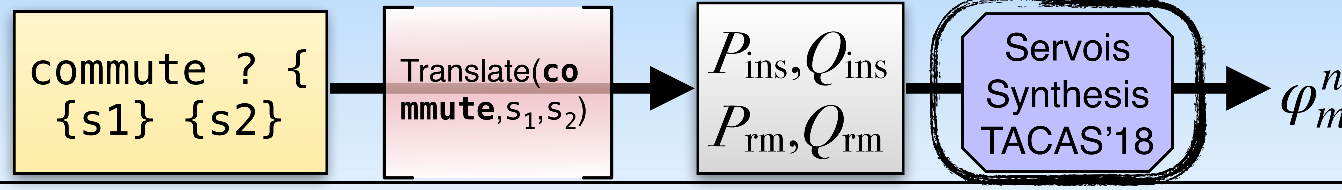
Logical ADT Specification

$$O = \left\{ \begin{array}{llllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



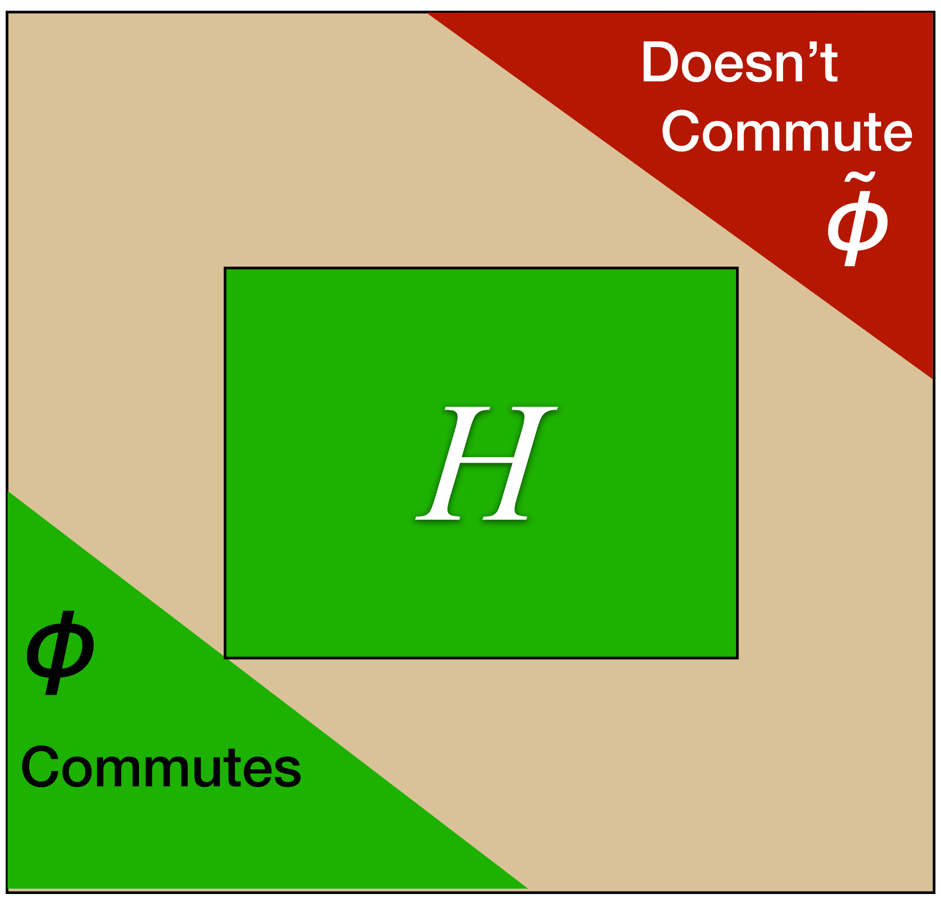
Refine(H)



Logical ADT Specification

$$O = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \longrightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



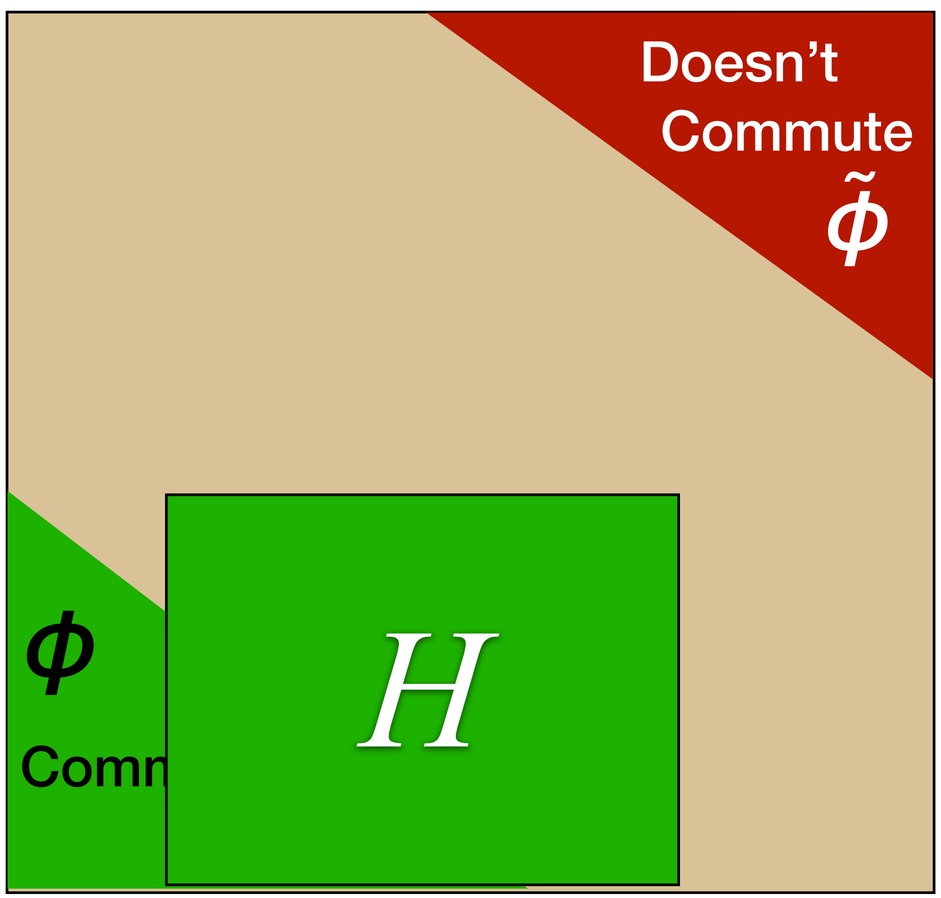
Refine(H)

✓ *If valid(H ⇒ m⋈n):* ***add H to ϕ***

Logical ADT Specification

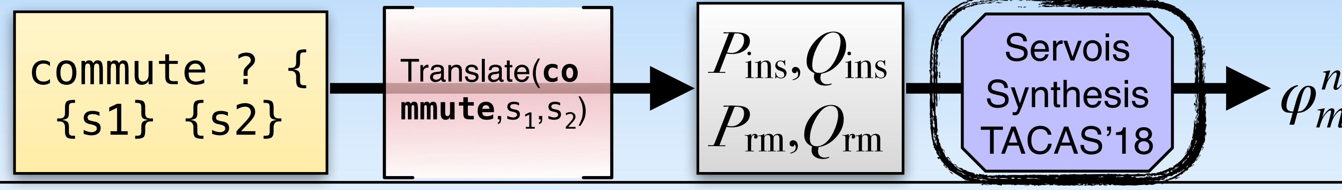
$$O = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



Refine(H)

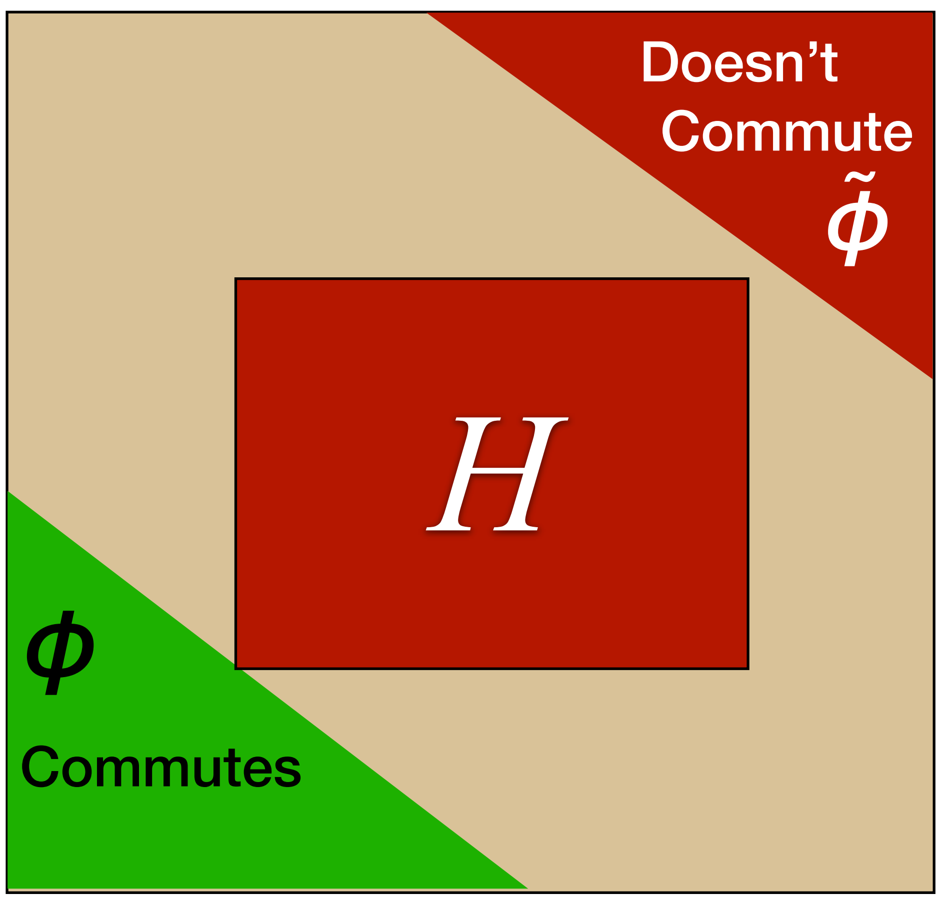
✓ *If valid(H ⇒ m⋈n):* ***add H to ϕ***



Logical ADT Specification

$$O = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \longrightarrow (P, Q) \end{array} \right\}$$

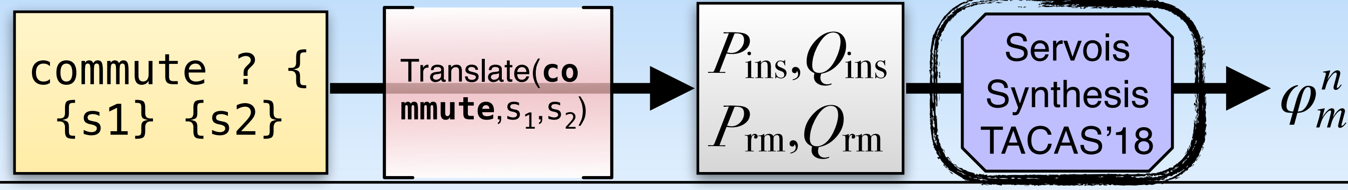
1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



Refine(H)

✓ *If valid(H ⇒ m⋈n):* **add H to ϕ**

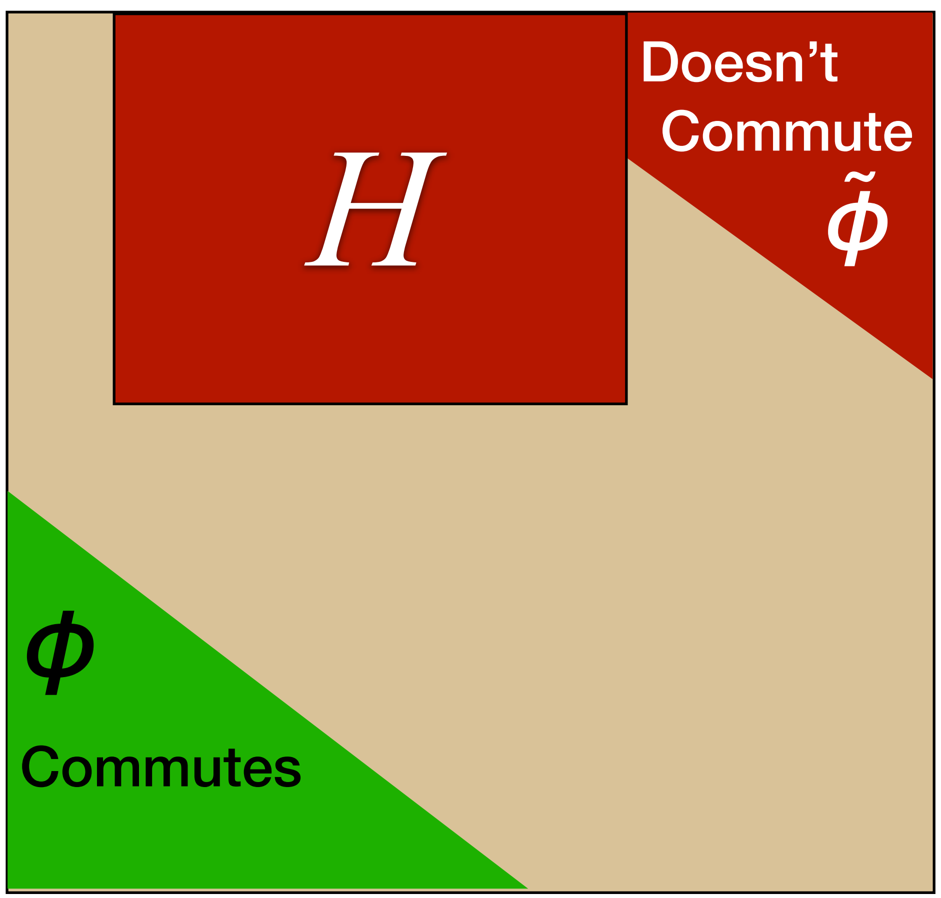
✓ *If valid(H ⇒ m⋈̸n):* **add H to $\tilde{\phi}$**



Logical ADT Specification

$$O = \left\{ \begin{array}{llllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \longrightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



Refine(H)

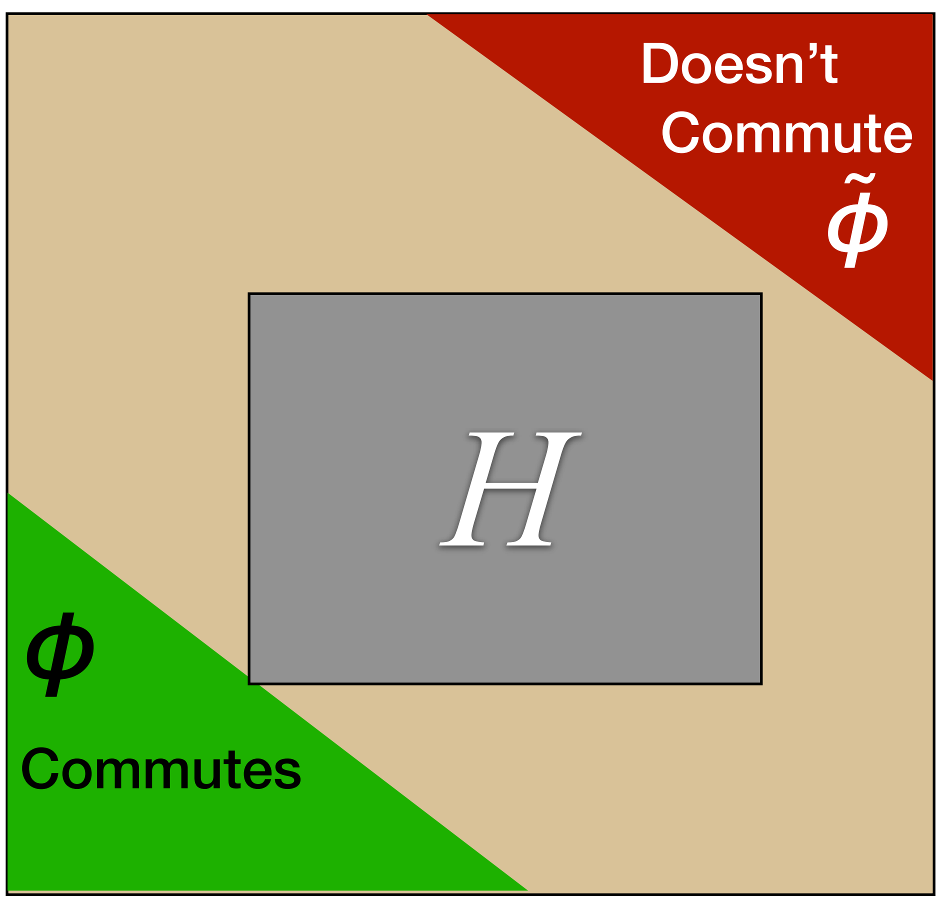
✓ *If valid(H ⇒ m⋈n):* **add H to ϕ**

✓ *If valid(H ⇒ m⋈̸n):* **add H to $\tilde{\phi}$**

Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{llll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} : \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} : \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

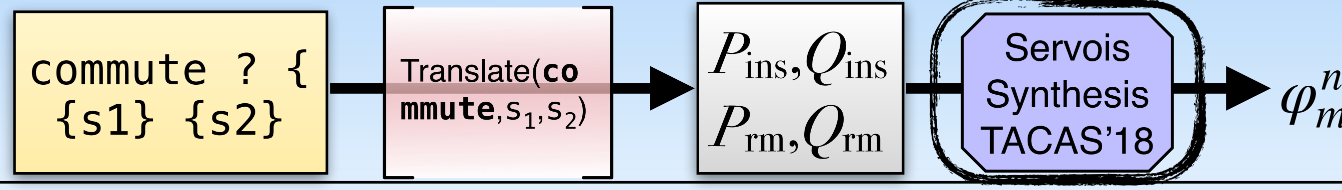
1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



Refine(H)

✓ *If valid(H ⇒ m⋈n):* **add H to ϕ**

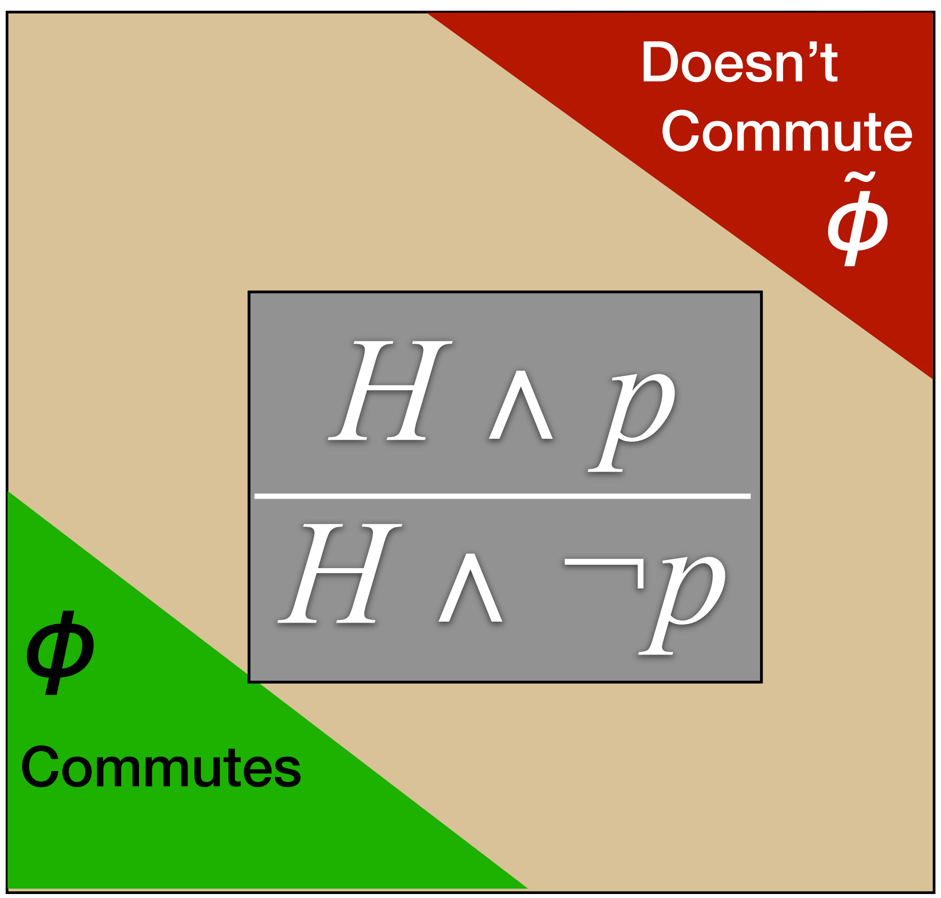
✓ *If valid(H ⇒ m⋈̸n):* **add H to $\tilde{\phi}$**



Logical ADT Specification

$$O = \left\{ \begin{array}{llllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



Refine(H)

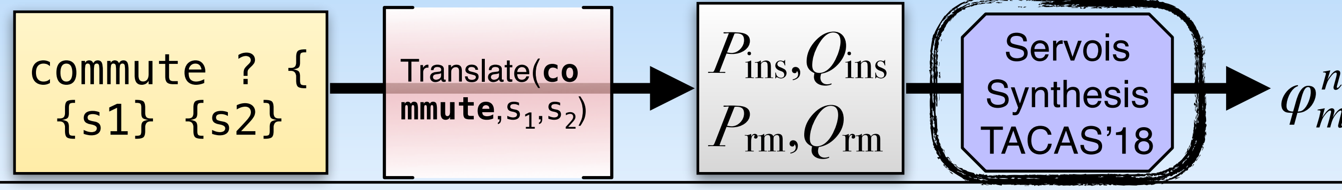
✓ *If valid(H ⇒ m⋈n):* ***add H to ϕ***

✓ *If valid(H ⇒ m⋈̸n):* ***add H to $\tilde{\phi}$***

✓ *If neither:*

P=CHOOSE(…)

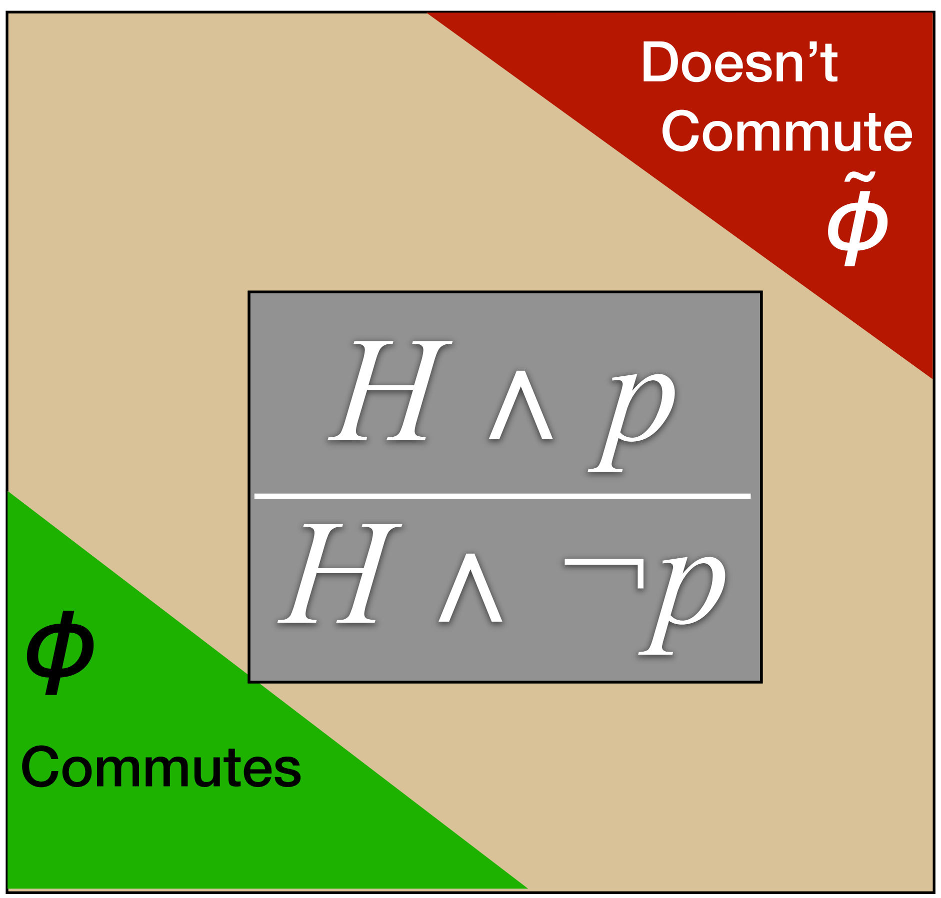
Refine(H ⋀ P); Refine(H ⋀ ¬P)



Logical ADT Specification

$$O = \left\{ \begin{array}{llll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \to (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi_m^n(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition via abstraction-refinement



Refine(H)

✓ *If valid(H ⇒ m⋈n):* **add H to ϕ**

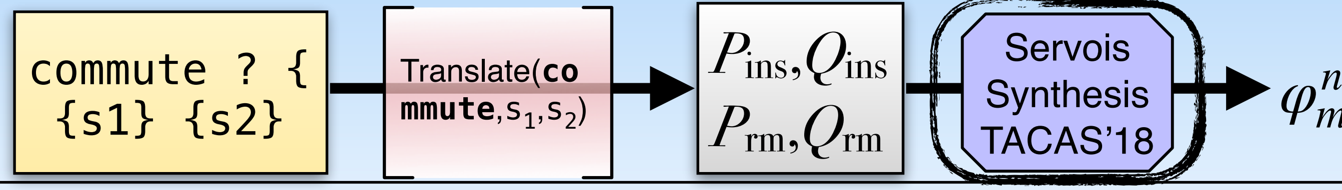
✓ *If valid(H ⇒ m⋈̸n):* **add H to $\tilde{\phi}$**

Use counterex's and "poke" heuristic. Generate preds, etc.

✓ *If neither:*

P=CHOOSE(…)

Refine(H ∧ P); Refine(H ∧ ¬P)

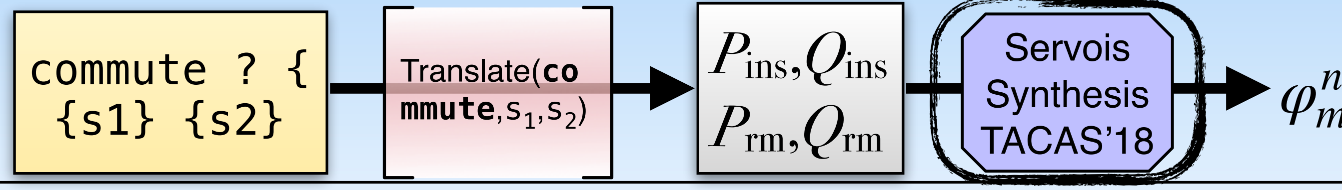


Logical ADT Specification

$$O = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

1. Verifying commute condition $\varphi^n_m(\sigma, \bar{x}, \bar{y})$
2. **Synthesize** commute condition

REFINE$^m_n(H, \mathcal{P})$ {
    if valid($H \Rightarrow m \,\hat{\bowtie}\, n$) then
        $\varphi := \varphi \vee H$;
    else if valid($H \Rightarrow m \,\not\hat{\bowtie}\, n$) then
        $\tilde{\varphi} := \tilde{\varphi} \vee H$;
    else
        let χ_c, χ_{nc} = counterex. to $\hat{\bowtie}$ and $\not\hat{\bowtie}$ (resp.) in
        let p = CHOOSE($H, \mathcal{P}, \chi_c, \chi_{nc}$) in
            REFINE$^m_n(H \wedge p, \mathcal{P} \setminus \{p\})$;
            REFINE$^m_n(H \wedge \neg p, \mathcal{P} \setminus \{p\})$;
}
main {
    φ := false; $\tilde{\varphi}$:= false;
    try { REFINE^m_n(true, $\mathcal{P}$); }
    catch (InterruptedException e) { skip; }
    return($\varphi, \tilde{\varphi}$);
}

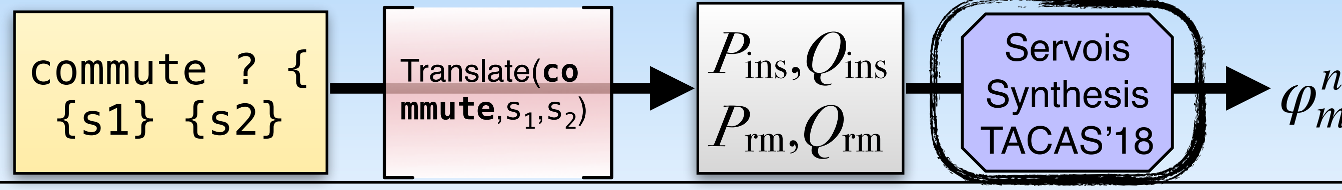


Set Abstract Data Type

S

`contains(`x`)/bool`, which performs a side-effect-free check whether the element x is in S; and

`add(`y`)/bool`, which adds y to S if it is not already in there and returns true, or otherwise returns false.

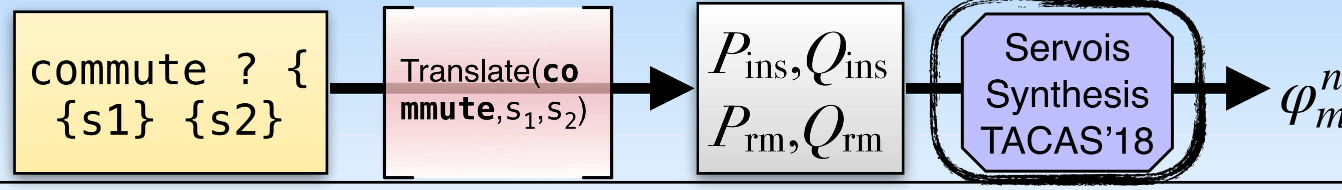

commute ? { {s1} {s2}
Translate(commute,s_1,s_2)
$P_{\text{ins}}, Q_{\text{ins}}$
$P_{\text{rm}}, Q_{\text{rm}}$
Servois Synthesis TACAS'18
φ_m^n

Don't Know

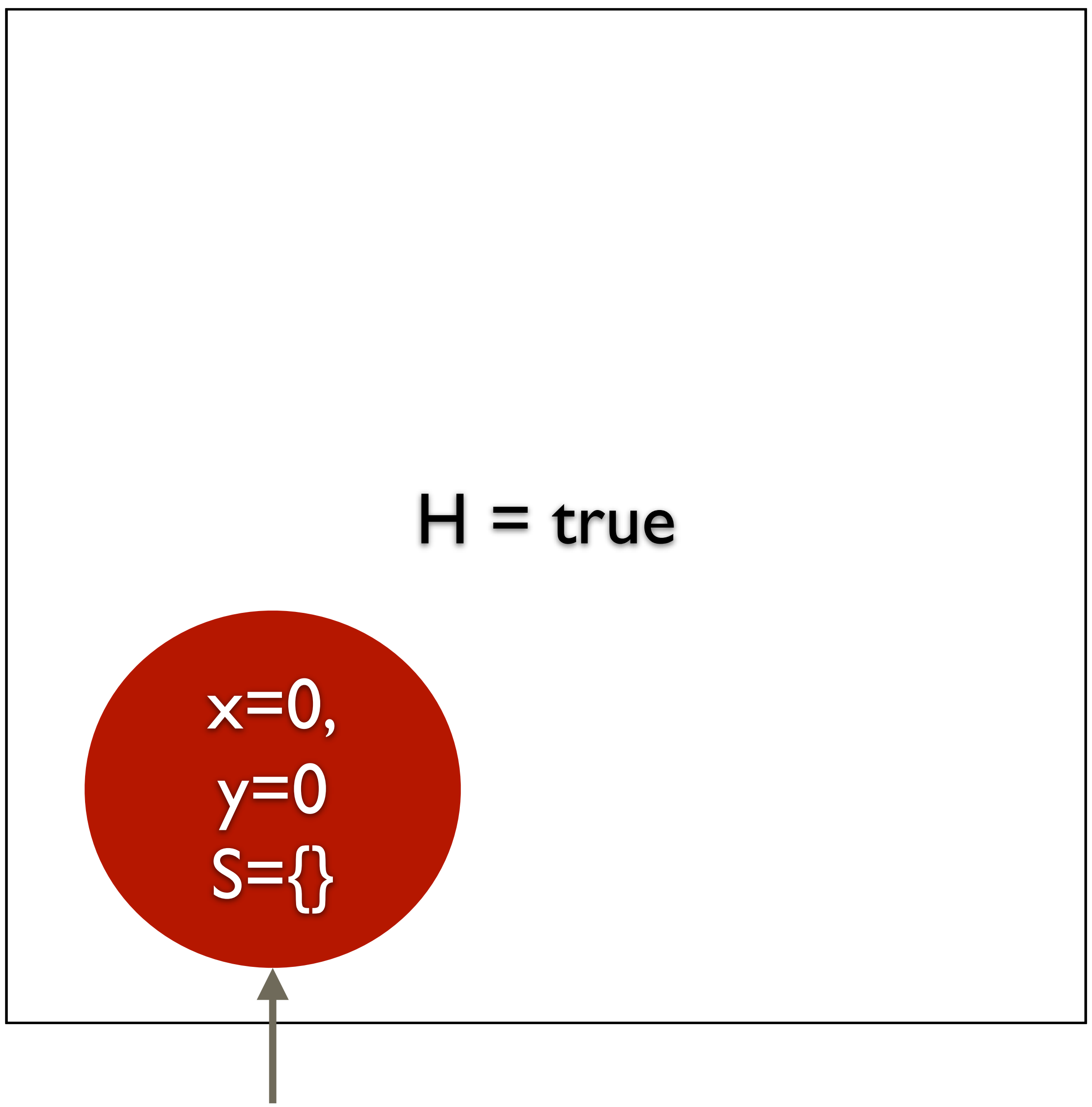
H = true

```
true =>
contains(x)/bool
⋈
add(y)/bool?
```

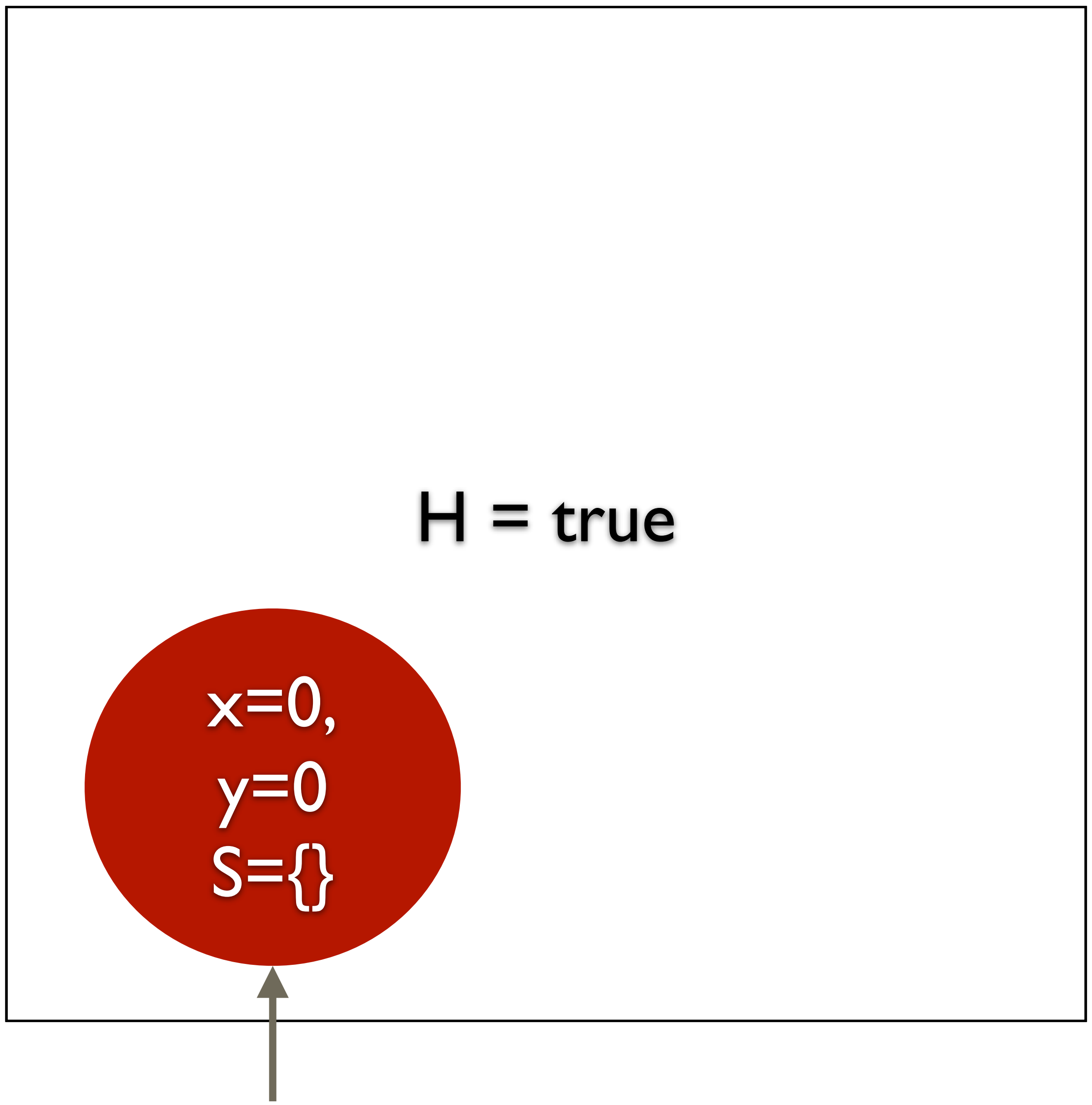
H = true




```
true =>
contains(x)/bool
⋈
add(y)/bool?
```



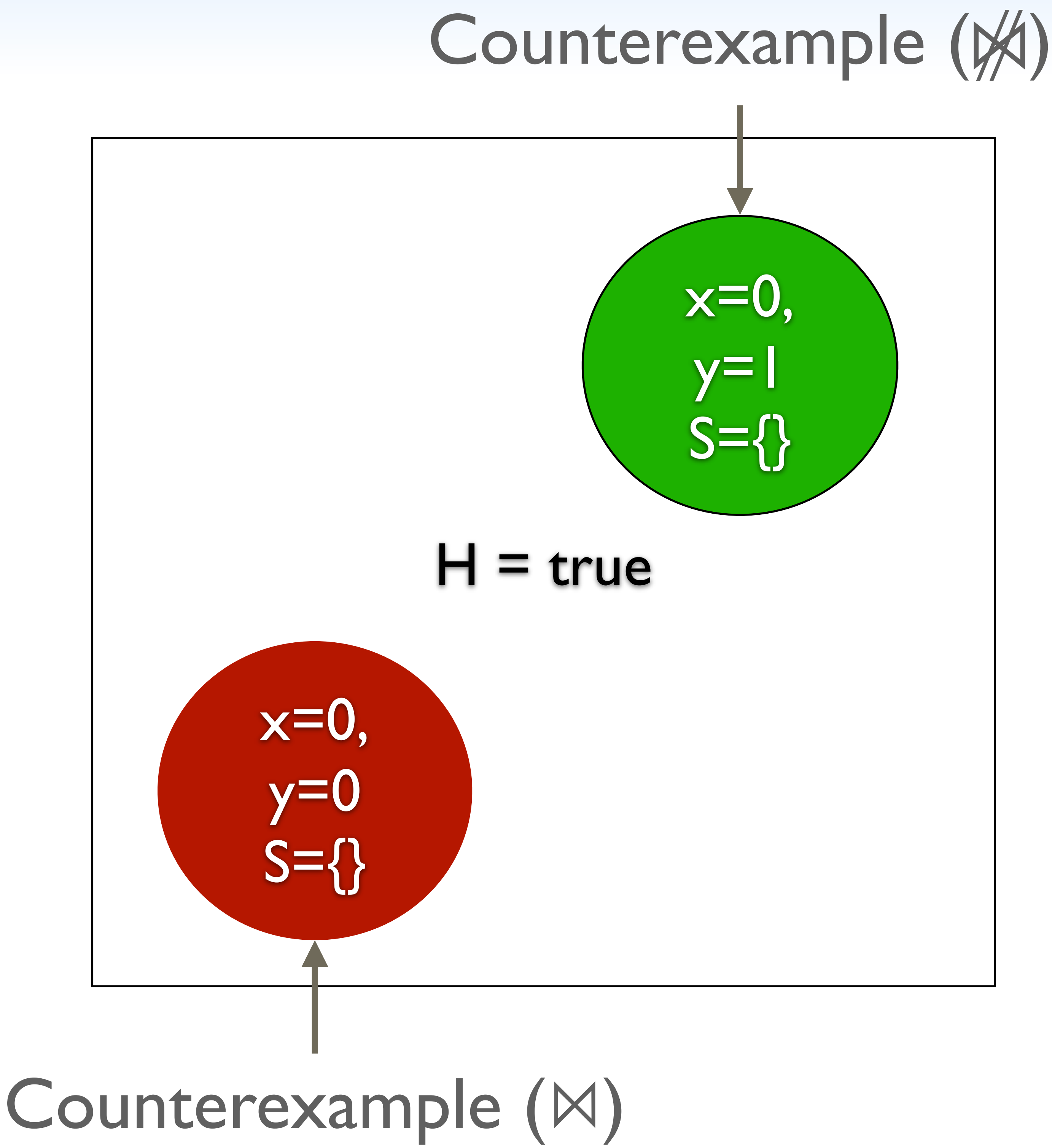
Counterexample (⋈)



Counterexample (⋈)

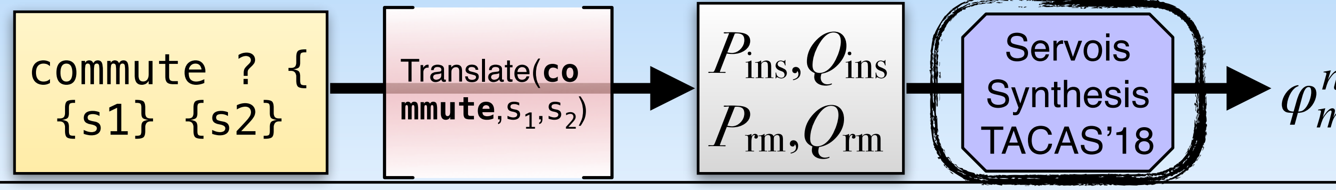
```
true =>
contains(x)/bool
⋈
add(y)/bool?
```

```
true =>
contains(x)/bool
⋈̸
add(y)/bool?
```

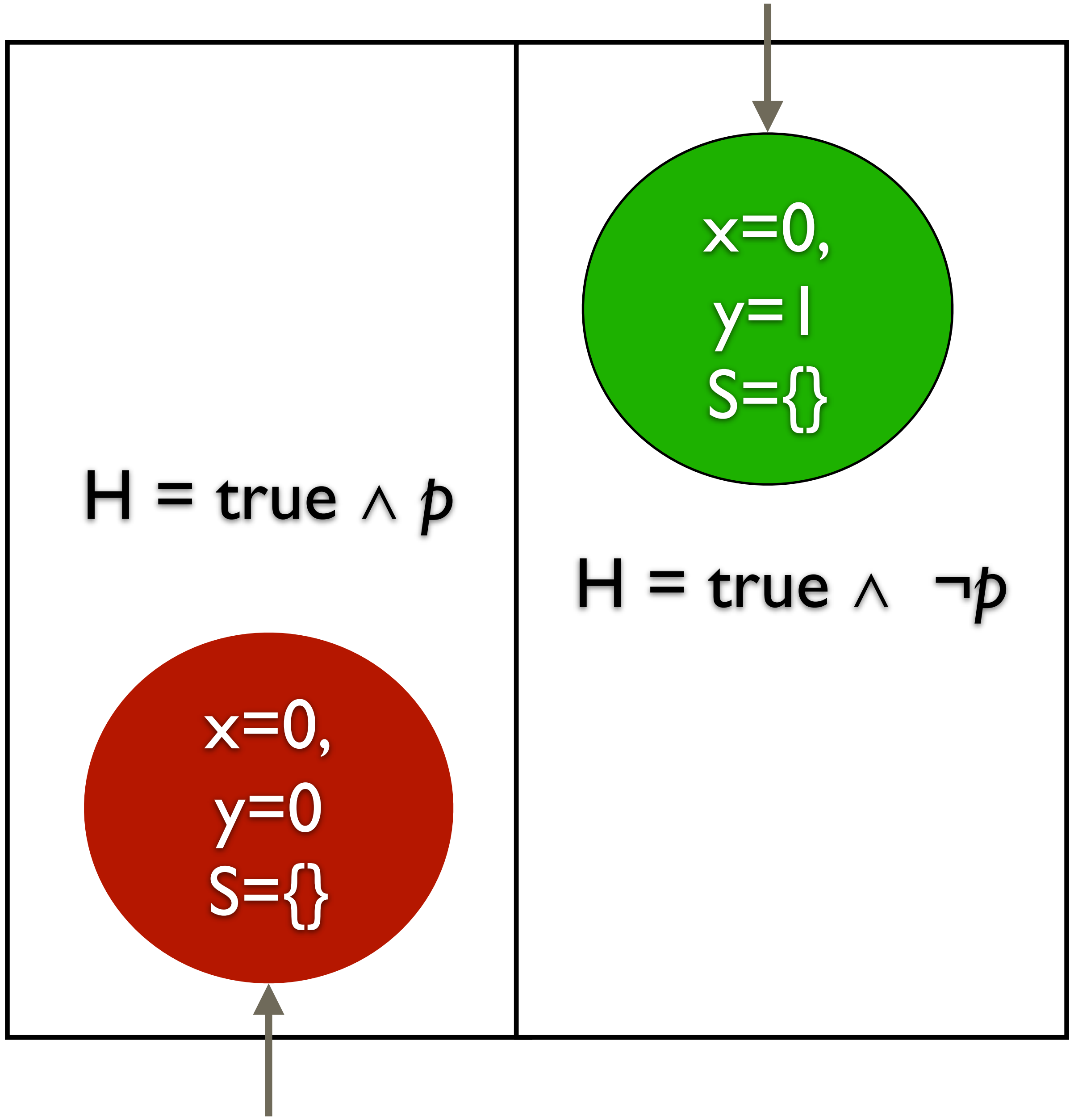


true =>
contains(x)/bool
⋈
add(y)/bool?

true =>
contains(x)/bool
⋈̸
add(y)/bool?



Counterexample (⋈̸)



Counterexample (⋈)

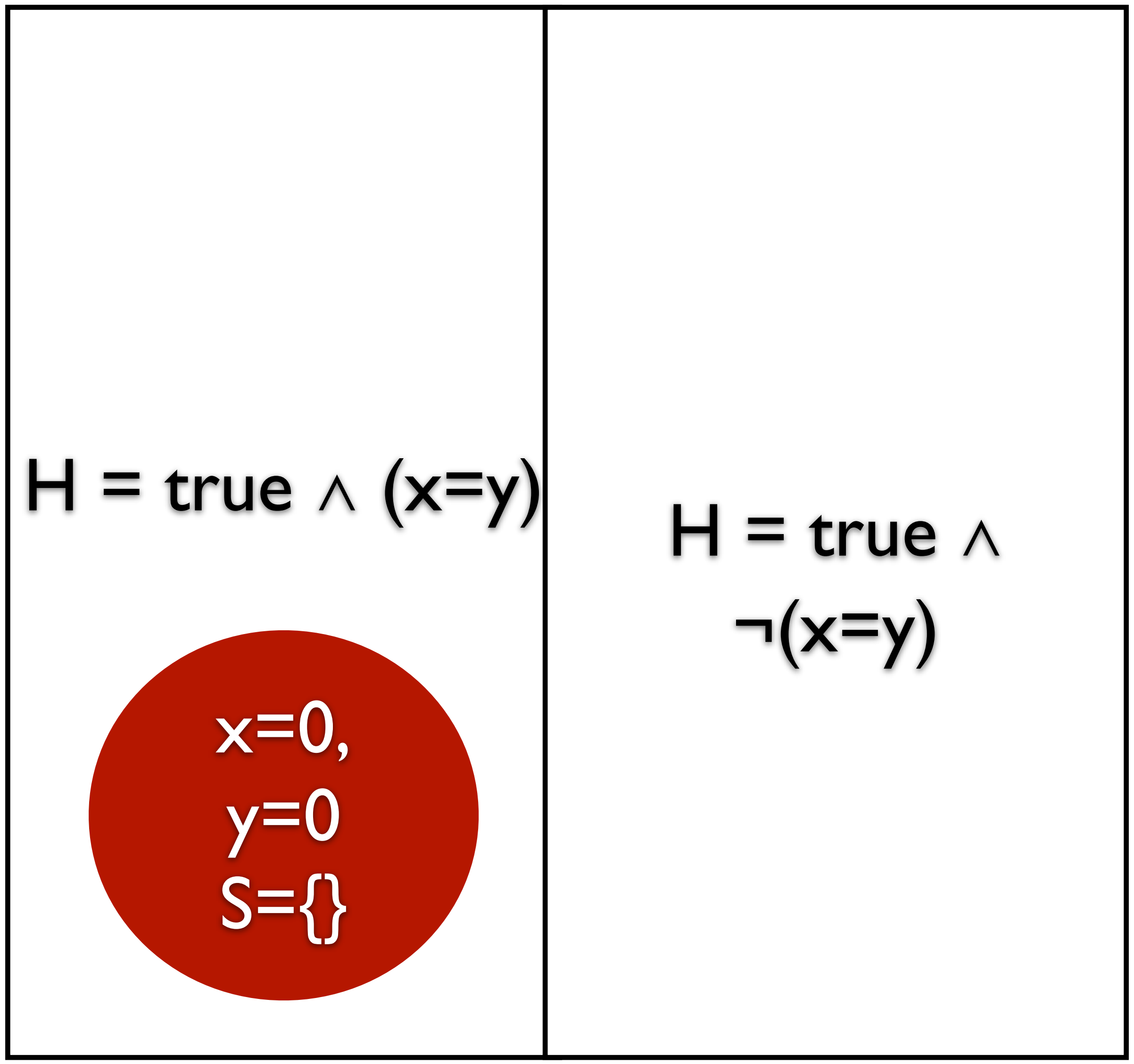
```
true =>
contains(x)/bool
⋈
add(y)/bool?
```

```
true =>
contains(x)/bool
⋈̸
add(y)/bool?
```

H = true ∧ (x=y)

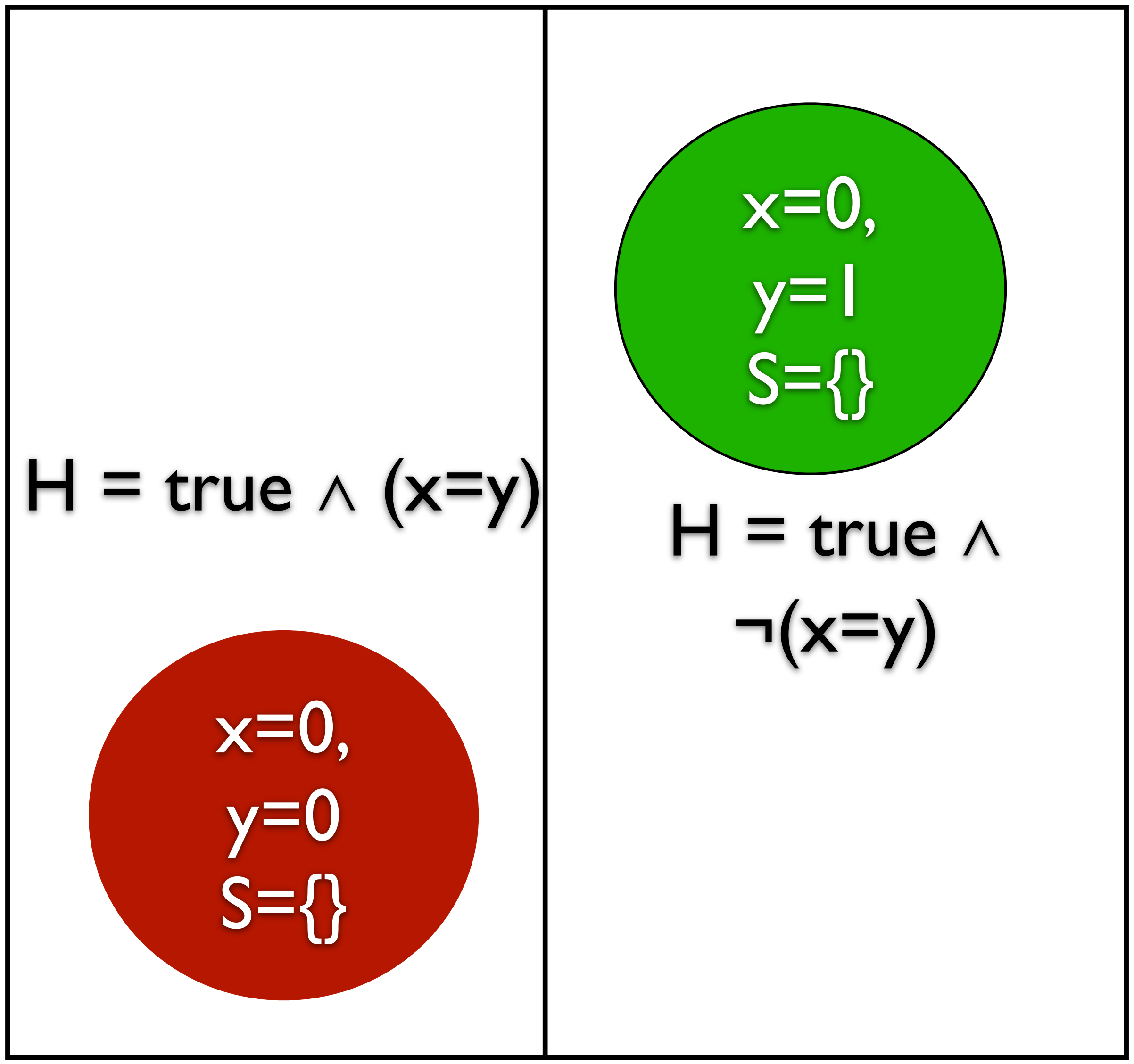
H = true ∧ ¬(x=y)

p: (x = y)

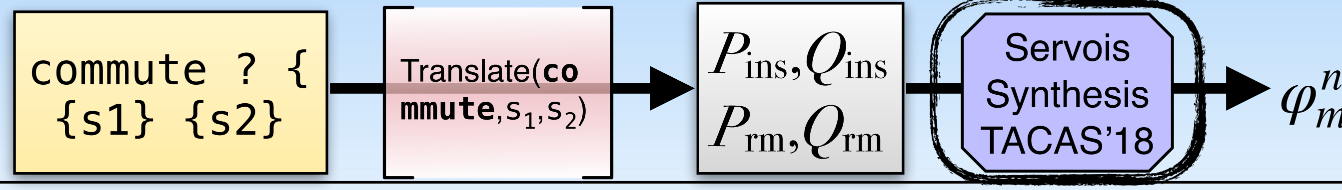


p: (x = y)

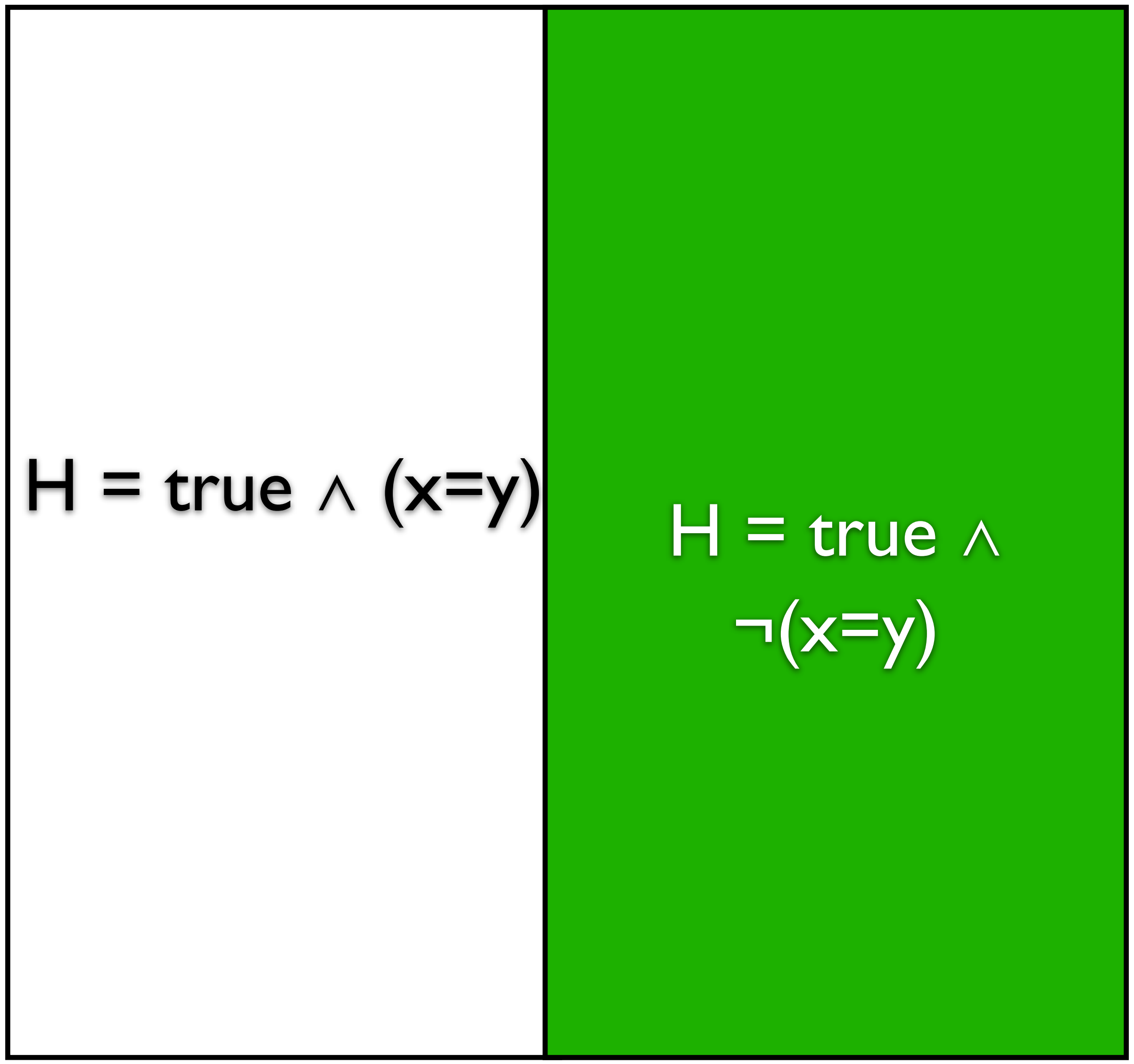
Part B: Commutativity Reasoning



p: (x = y)



H: ¬(x = y)



H =>
`contains(x)/bool`
⋈
`add(y)/bool?`

Yes!

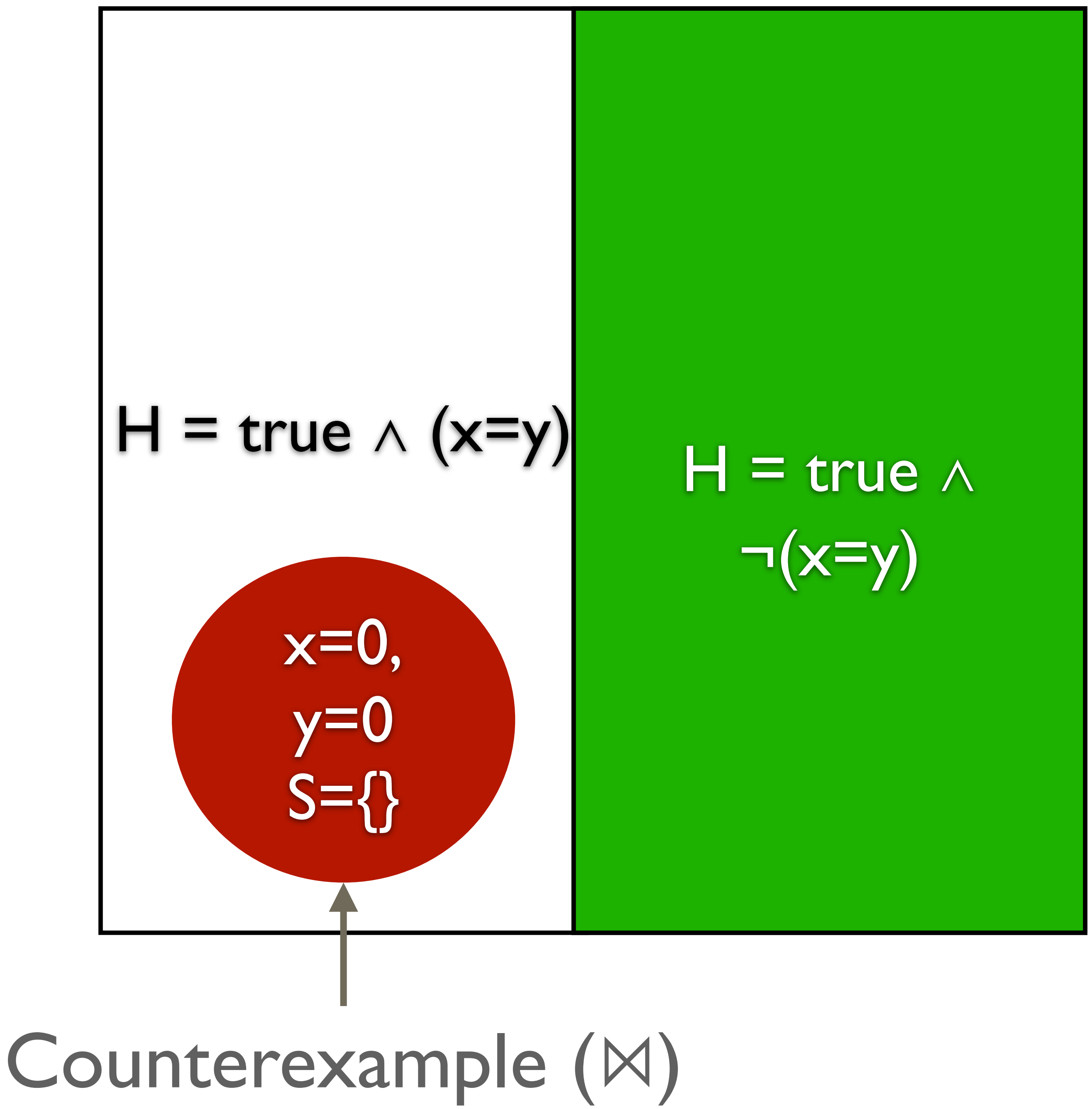
H: (x = y)

H = true ∧ (x=y)	H = true ∧ ¬(x=y)

H =>
`contains(x)/bool`
⋈
`add(y)/bool?`

H =>
`contains(x)/bool`
⋈̸
`add(y)/bool?`

H: (x = y)

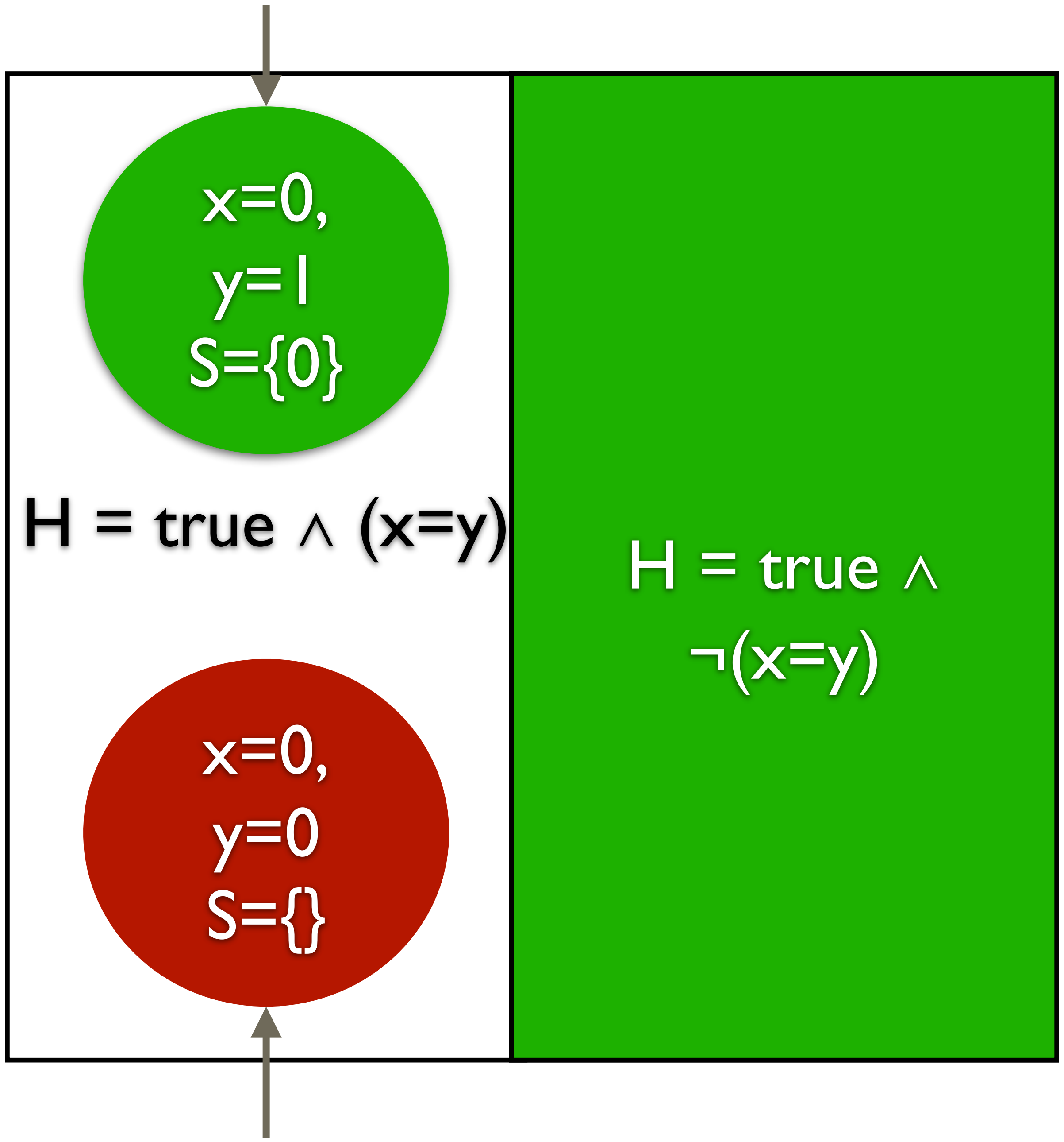


H =>
`contains(x)/bool`
⋈
`add(y)/bool?`

H =>
`contains(x)/bool`
⋈̸
`add(y)/bool?`

Counterexample (⋈̸)

H: (x = y)



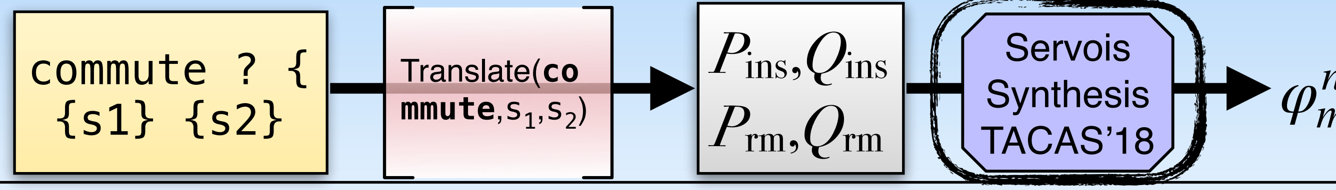
Counterexample (⋈)

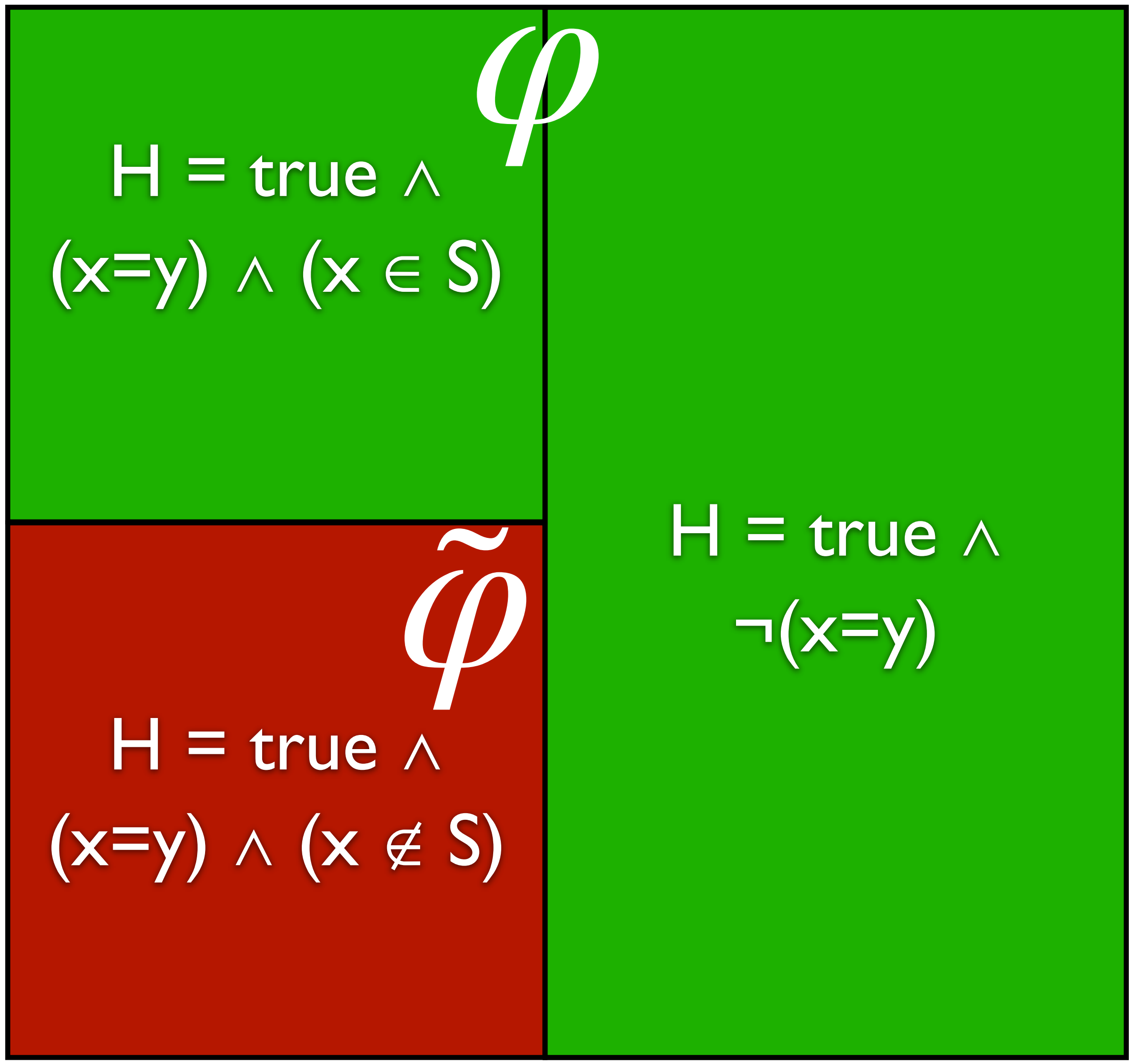
H =>
`contains(x)/bool`
⋈
`add(y)/bool?`

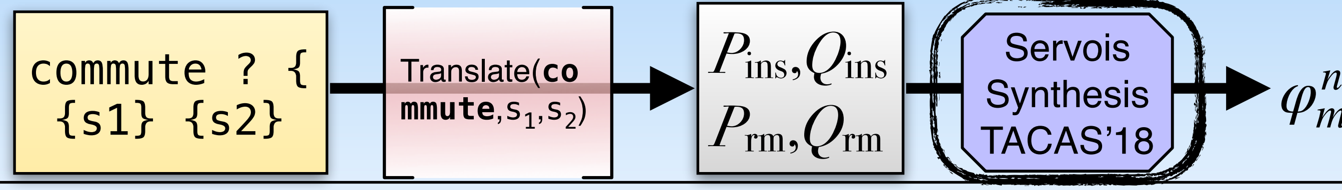
H =>
`contains(x)/bool`
⋈̸
`add(y)/bool?`

H = true ∧ (x=y) ∧ (x ∈ S)	H = true ∧ ¬(x=y)
H = true ∧ (x=y) ∧ (x ∉ S)	

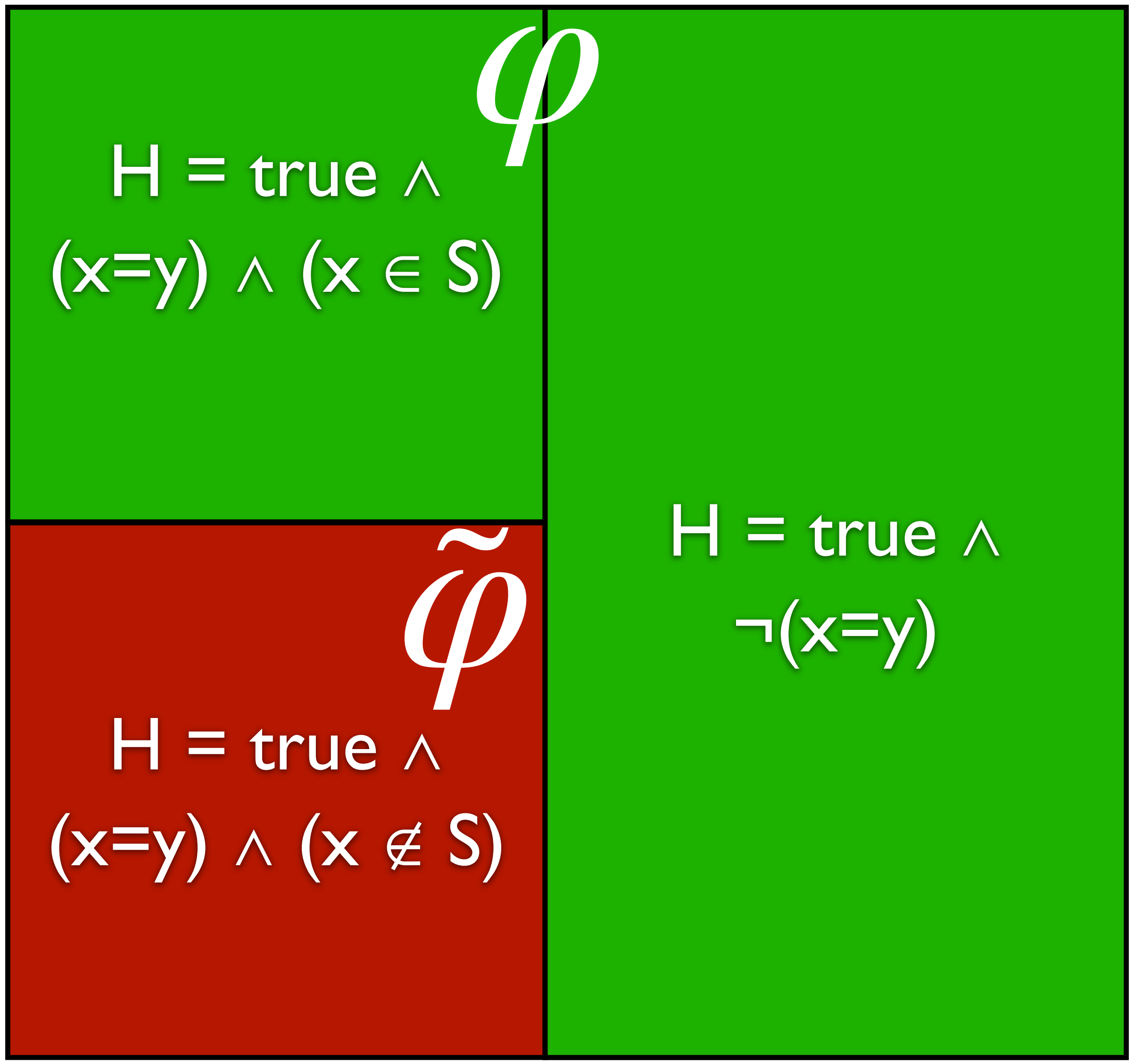
p': (x ∈ S)


commute ? { {s1} {s2}
Translate(commute,s_1,s_2)
P_{ins}, Q_{ins}
P_{rm}, Q_{rm}
Servois Synthesis TACAS'18
φ_m^n

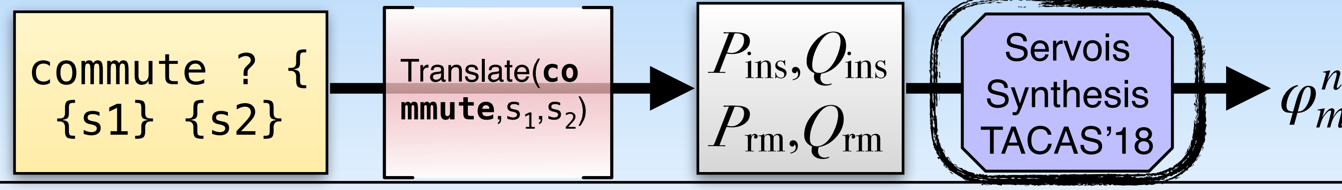

φ
H = true ∧ (x=y) ∧ (x ∈ S)
$\tilde{\varphi}$
H = true ∧ (x=y) ∧ (x ∉ S)
H = true ∧ ¬(x=y)



Part B: Commutativity Reasoning



$$\varphi \equiv x \neq y \vee (x = y \wedge x \in S)$$



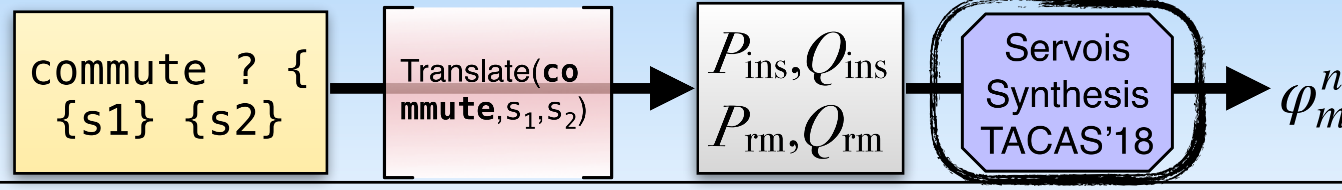
Part B: Commutativity Reasoning

	$m(\bar{x})$		$n(\bar{y})$	**Simple** Qs (time)	**Poke** Qs (time)	φ_n^m **(Poke)**
Counter	decrement	⋈	decrement	3 (0.1)	3 (0.1)	true
	increment	▷	decrement	10 (0.3)	34 (0.9)	$\neg(0 = c)$
	decrement	▷	increment	3 (0.1)	3 (0.1)	true
	decrement	⋈	reset	2 (0.1)	2 (0.1)	false
	decrement	⋈	zero	6 (0.1)	26 (0.6)	$\neg(1 = c)$
	increment	⋈	increment	3 (0.1)	3 (0.1)	true
	increment	⋈	reset	2 (0.0)	2 (0.1)	false
	increment	⋈	zero	10 (0.3)	34 (0.8)	$\neg(0 = c)$
	reset	⋈	reset	3 (0.1)	3 (0.1)	true
	reset	⋈	zero	9 (0.2)	30 (0.6)	$0 = c$
	zero	⋈	zero	3 (0.1)	3 (0.1)	true
Acum.	increase	⋈	increase	3 (0.1)	3 (0.1)	true
	increase	⋈	read	13 (0.3)	28 (0.6)	$c + x_1 = c$
	read	⋈	read	3 (0.0)	3 (0.0)	true
Set	add	⋈	add	10 (0.4)	140 (4.4)	$(y_1 = x_1 \wedge y_1 \in S) \vee \neg(y_1 = x_1)$
	add	⋈	contains	10 (0.4)	122 (3.6)	$x_1 \in S \vee (\neg(x_1 \in S) \wedge \neg(y_1 = x_1))$
	add	⋈	getsize	6 (0.2)	31 (0.9)	$x_1 \in S$
	add	⋈	remove	6 (0.2)	66 (2.2)	$\neg(y_1 = x_1)$
	contains	⋈	contains	3 (0.1)	3 (0.1)	true
	contains	⋈	getsize	3 (0.1)	3 (0.1)	true
	contains	⋈	remove	17 (0.5)	160 (4.8)	$S \setminus \{x_1\} = \{y_1\} \vee (\ldots \wedge y_1 \in \{x_1\}) \vee$ …
	getsize	⋈	getsize	3 (0.1)	3 (0.1)	true
	getsize	⋈	remove	13 (0.3)	37 (1.0)	$\neg(y_1 \in S)$

commute ? { {s1} {s2} → Translate(commute, s_1, s_2) → P_{ins}, Q_{ins} P_{rm}, Q_{rm} → Servois Synthesis TACAS'18 → φ_m^n

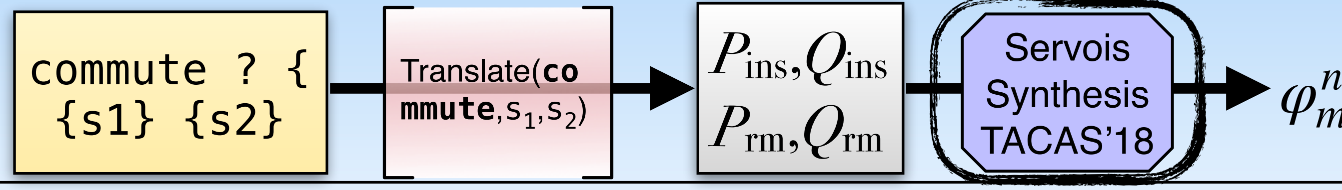
HashTable

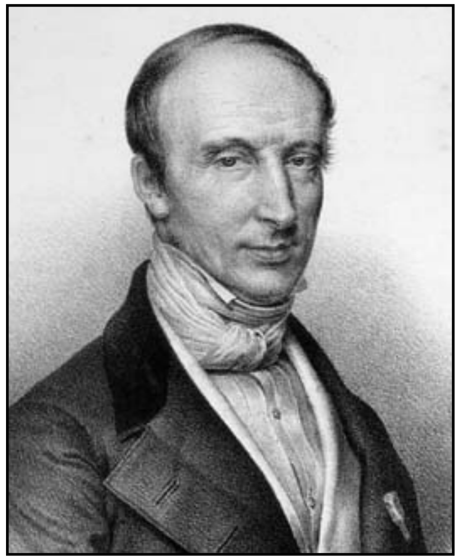
$m(\bar{x})$		$n(\bar{y})$	**Simple** Qs (time)	**Poke** Qs (time)	φ_n^m **(Poke)**
decrement	⋈	decrement	3 (0.1)	3 (0.1)	true
increment	▷	decrement	10 (0.3)	34 (0.9)	$\neg(0 = c)$
get	⋈	get	3 (0.1)	3 (0.1)	true
get	⋈	haskey	3 (0.1)	3 (0.1)	true
put	▷	get	13 (0.4)	74 (2.3)	$(H[x_1 \leftarrow x_2] = H \wedge y_1 \in keys)$ $\vee(\neg(H[x_1 \leftarrow x_2] = H) \wedge \neg(y_1 = x_1))$
get	▷	put	10 (0.3)	48 (1.5)	$[H[y_1] = y_2] \vee [\neg(H[y_1] = y_2) \wedge \neg(y_1 = x_1)]$
remove	▷	get	3 (0.1)	3 (0.1)	true
get	▷	remove	13 (0.4)	40 (1.2)	$\neg(y_1 = x_1)$
get	⋈	size	3 (0.1)	3 (0.1)	true
haskey	⋈	haskey	3 (0.1)	3 (0.1)	true
haskey	⋈	put	10 (0.3)	52 (1.6)	$[y_1 \in keys] \vee [\neg(y_1 \in keys) \wedge \neg(y_1 = x_1)]$
haskey	⋈	remove	17 (0.5)	44 (1.3)	$[x_1 \in keys \wedge \neg(y_1 = x_1)] \vee [\neg(x_1 \in keys)]$
haskey	⋈	size	3 (0.1)	3 (0.1)	true
put	⋈	put	24 (0.9)	357 (13.5)	$\ldots \vee (\neg(H[y_1] = y_2) \wedge \neg(y_1 = x_1))$
put	⋈	remove	6 (0.3)	33 (1.2)	$\neg(y_1 = x_1)$
put	⋈	size	6 (0.2)	23 (0.8)	$x_1 \in keys$
remove	⋈	remove	21 (0.8)	192 (6.9)	$[keys \setminus \{x_1\} = \{y_1\}] \vee [\ldots]$
contains	⋈	remove	[illegible]	[illegible]	$S \setminus \{x_1\} = \{y_1\} \vee (\ldots \wedge y_1 \in \{x_1\}) \vee$ ...
getsize	⋈	getsize	3 (0.1)	3 (0.1)	true
getsize	⋈	remove	13 (0.3)	37 (1.0)	$\neg(y_1 \in S)$



Applications *of Commutativity Synthesis*

- **Smart Contracts**. Ensure determinism.
- **Concurrent verification**. Partial Order reduction, transactional memory, etc.
- **Testing for interactions between code blocks**.
- **CRDTs**. Distributed computing.
- **Refactoring (and other relational reasoning)**.
- **Code synthesis**. Eg, synthesized conditions become specification for synchronization synthesis.
- **Commute blocks in Veracity!**





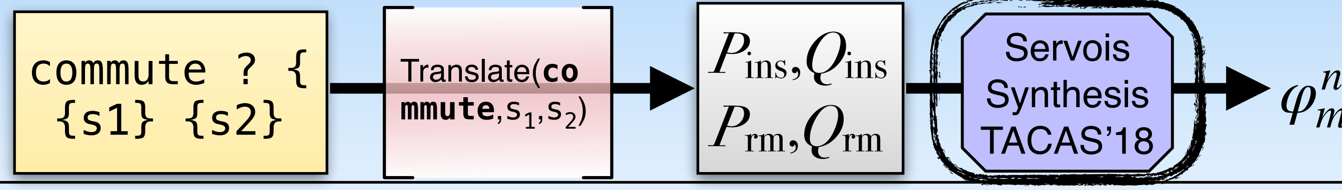
SERVOIS

Implemented in Python with CVC4.

Available on GitHub.

Kshitij Bansal
PhD student at NYU
Now at Google

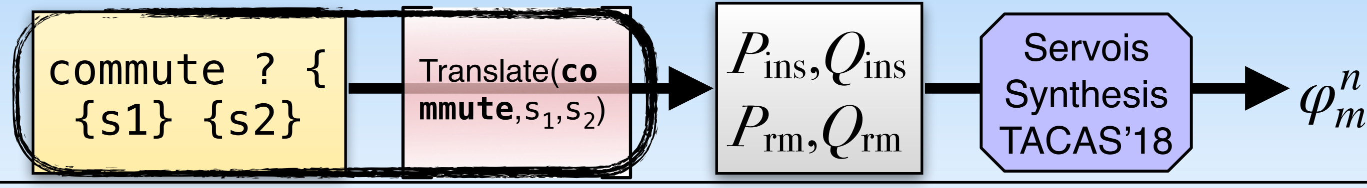
Omer Tripp
PhD student at TAU
Now at Amazon

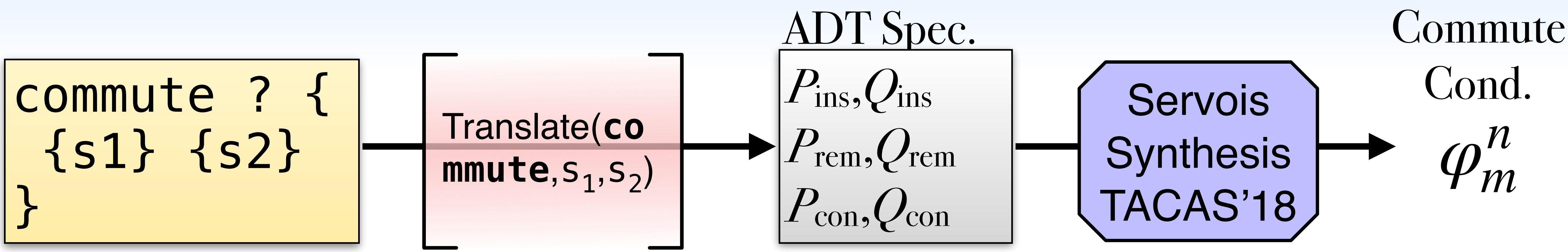


SERVOIS 2.0

Coming very soon!

- **More solvers!** CVC4, CVC5, Z3, …
- **More theories!** e.g bitvectors.
- **Faster!** Reimplemented in OCaml from scratch.
- **Better predicate generation**.
- **Better predicate selection**.
- **Command-line or Library API**.





Veracity: Declarative Multicore Programming with Commutativity

ADAM CHEN, Stevens Institute of Technology, USA
PARISA FATHOLOLUMI, Stevens Institute of Technology, USA
ERIC KOSKINEN, Stevens Institute of Technology, USA
JARED PINCUS, Stevens Institute of Technology, USA

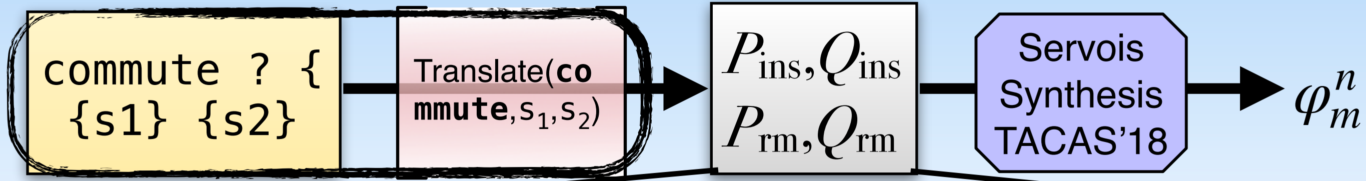
There is an ongoing effort to provide programming abstractions that ease the burden of exploiting multicore hardware. Many programming abstractions (*e.g.*, concurrent objects, transactional memory, etc.) simplify matters, but still involve intricate engineering. We argue that some difficulty of multicore programming can be meliorated through a declarative programming style in which programmers directly express the independence of fragments of sequential programs.

In our proposed paradigm, programmers write programs in a familiar, sequential manner, with the added ability to explicitly express the conditions under which code fragments sequentially commute. Putting such commutativity conditions into source code offers a new entry point for a compiler to exploit the known connection between commutativity and parallelism. We give a semantics for the programmer's sequential perspective and, under a correctness condition, find that a compiler-transformed parallel execution is equivalent to the sequential semantics. Serializability/linearizability are not the right fit for this condition, so we introduce scoped serializability and show how it can be enforced with lock synthesis techniques.

We next describe a technique for automatically verifying and synthesizing commute conditions via a new reduction from our commute blocks to logical specifications, upon which symbolic commutativity reasoning can be performed. We implemented our work in a new language called Veracity, implemented in Multicore OCaml. We show that commutativity conditions can be automatically generated across a variety of new benchmark programs, confirm the expectation that concurrency speedups can be seen as the computation increases, and apply our work to a small in-memory filesystem and an adaptation of a crowdfund blockchain smart contract.

1 INTRODUCTION

Writing concurrent programs is difficult. Researchers and practitioners, seeking to make life easier,



Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{lll} \text{state} & : & (Var, Type)\text{list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; \end{array} \quad \begin{array}{lll} \text{methods} & : & \text{Meth list}; \\ \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

m_1 :

```
x = calc1(a);
c = c + (x*x);
```

m_2 :

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

`commute ? { {s1} {s2}` → Translate(`commute`, s_1, s_2) → P_{ins}, Q_{ins} / P_{rm}, Q_{rm} → Servois Synthesis TACAS'18 → φ_m^n

Logical ADT Specification

$$O = \left\{ \begin{array}{lllllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

m_1 :

```
x = calc1(a);
c = c + (x*x);
```

But m_1 is code, not a logical spec.

m_2 :

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

```
commute ? {
 {s1} {s2}
```

Translate(commute, s_1, s_2) → P_{ins}, Q_{ins} P_{rm}, Q_{rm} → Servois Synthesis TACAS'18 → φ^n_m

Part B: Commutativity Reasoning

Logical ADT Specification

$$O = \left\{ \begin{array}{lllllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

m_1 :

```
x = calc1(a);
c = c + (x*x);
```

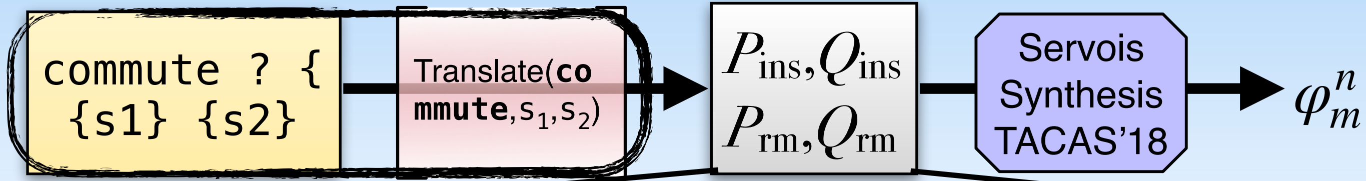
But m_1 is code, not a logical spec.

m_2 :

```
if (c > 0 && y < 0) {
   c = c - 1;
   z = calc2(y);
} else {
   z = calc3(y);
}
```

Translate to a logical post-condition.

```
(or (and
    (let ((x_1 a)
    (let ((c_1 (+ c (* x_1 x_1))))
    (and (= c_new c_1) (= x_new x_1))))
(= a_new a) (= z_new z) (= y_new y)
(= size_new size))))
```



Logical ADT Specification

$$\mathcal{O} = \left\{ \begin{array}{llllll} \text{state} & : & (Var, Type)\text{list}; & \text{methods} & : & \text{Meth list}; \\ \text{eq} & \subseteq & \text{state} \times \text{state}; & \text{spec} & : & \text{Meth} \rightarrow (P, Q) \end{array} \right\}$$

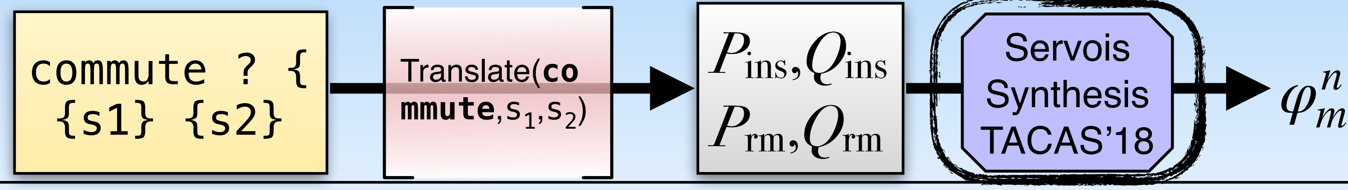
Translation

- **Nested commute statements?** Treat them as sequential composition!

$$Tr(\textbf{commute}\ c\ s_1\ s_2) = Tr(s_1;s_2)$$

- **Built-in ADTs?**

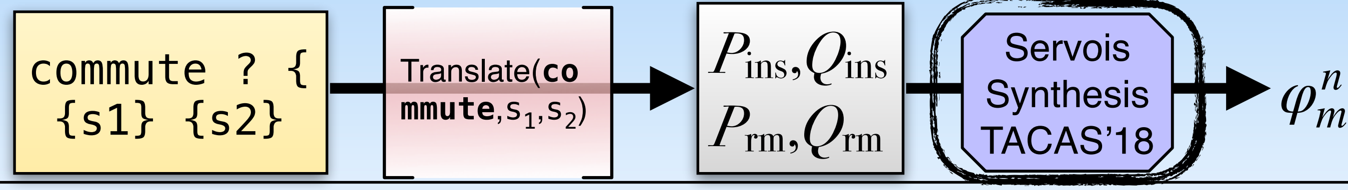
$$Tr(\textbf{tbl[}e_1\textbf{]} \textbf{ = } e_2) = inlineSpec(HT, \textbf{tbl}, \ldots)$$




```
int main(int argc, string[] argv) {
    hashtable[int,int] tbl = new hashtable[int,int];
    int n = int_of_string(argv[1]);
    int x = int_of_string(argv[2]);
    int y = int_of_string(argv[3]);
    int z = int_of_string(argv[4]);
    tbl[x] = 42;
    tbl[z] = 42;
    commute _ {
        {
            calc1(n);
            if(ht_mem(tbl, x))
                y = tbl[x];
        }
        {   if(ht_mem(tbl, z)) {
                y = tbl[z];
            }
            calc2(n);}
    }
    return 0;
}
```

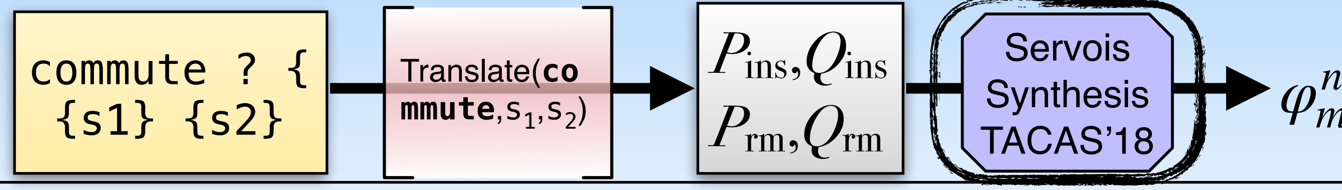
```
ejk@arran:veracity/src$ ./vcy.exe infer ../benchmarks/ht-cond-mem-get.vcy
Inferred condition at ../benchmarks/ht-cond-mem-get.vcy:
  [11.2-22.3]: tbl[x] == tbl[z] && !(x == z) || x == z

ejk@arran:veracity/src$
```

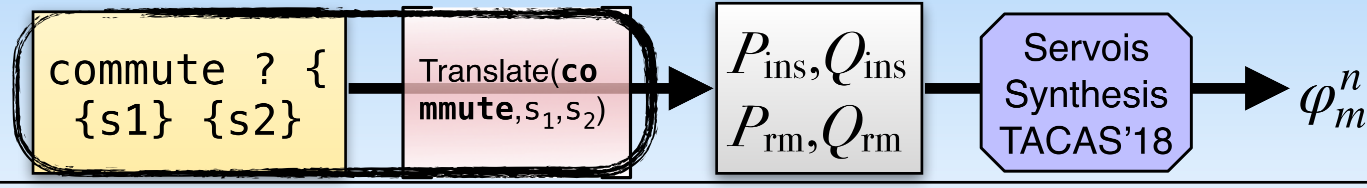


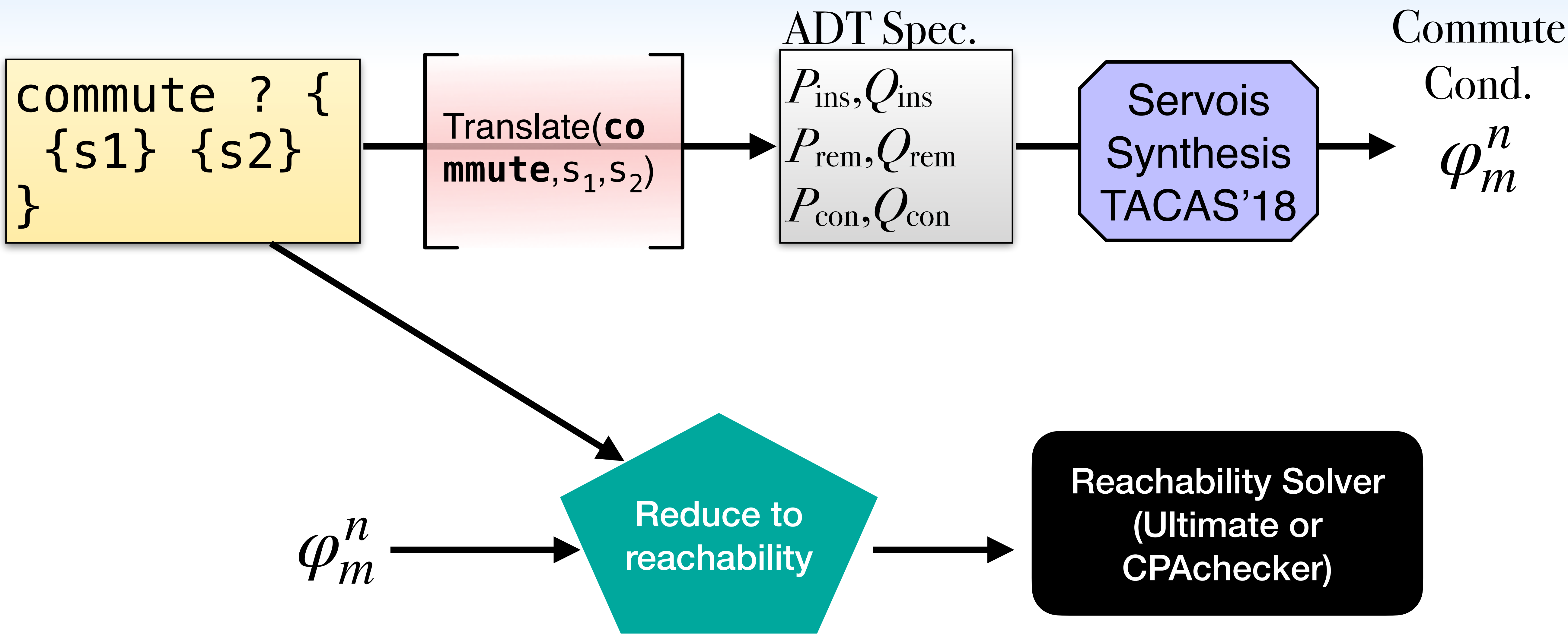
Part B: Commutativity Reasoning

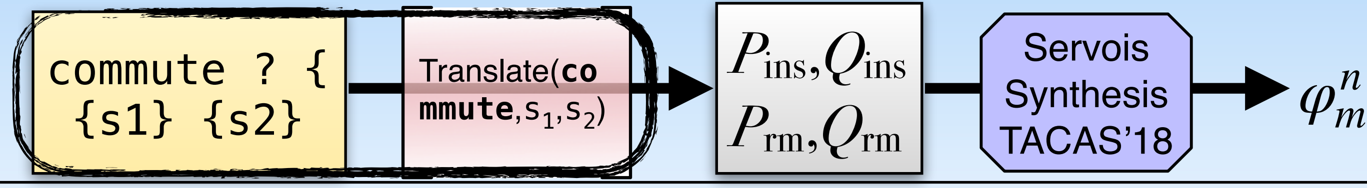
<table>
<tr><th colspan="3">Group 1: Automatically Inferred Commute Conditions. All benchmarks, except those below in group (3).</th></tr>
<tr><th>Program</th><th>Time (s)</th><th>Inferred Conditions</th></tr>
<tr><td>array-disjoint</td><td>0.63</td><td>i != j && x != y || x == y</td></tr>
<tr><td>array1</td><td>0.75</td><td>1 != r[0] && r[0] + 1 != y && r[0] <= 1 || r[0] + 1 == y && r[0] <= 1</td></tr>
<tr><td>array2</td><td>1.11</td><td>0 > a[0] && 1 != x || 1 == x</td></tr>
<tr><td>array3</td><td>1.13</td><td>d != e && a != b || a == b</td></tr>
<tr><td>calc</td><td>⏱</td><td>1 == y && 0 != y && 1 > c && 1 != c || ... || 1 == c</td></tr>
<tr><td>conditional</td><td>0.18</td><td>x > 0</td></tr>
<tr><td>counter</td><td>0.20</td><td>0 != c</td></tr>
<tr><td>dict</td><td>3.82</td><td>i != r && c + x != y || c + x == y</td></tr>
<tr><td>dot-product</td><td>0.24</td><td>true</td></tr>
<tr><td>even-odd</td><td>1.18</td><td>x % 2 == x + y && 0 != y || 0 == y</td></tr>
<tr><td>ht-add-put</td><td>2.24</td><td>tbl[z] == u + 1 && u + 1 != z</td></tr>
<tr><td>ht-cond-mem-get</td><td>1.54</td><td>tbl[x] == tbl[z] && x != z || x == z</td></tr>
<tr><td>ht-cond-size-get</td><td>0.83</td><td>ht_size(tbl) <= 0 && 0 != z || 0 == z</td></tr>
<tr><td>ht-simple</td><td>30.64</td><td>x + a != z && 3 == tbl[z] && y != z</td></tr>
<tr><td>linear-bool</td><td>3.62</td><td>0 <= y && 3 == x && 2 != x && 1 != x && x > 0 && 0 != x || 0 > y + 3 * x && 2 == x && 1 != x && x > 0 && 0 != x</td></tr>
<tr><td>linear-cond</td><td>2.65</td><td>2 <= y && 2 != y && 1 != y || ... || 1 == y</td></tr>
<tr><td>linear</td><td>0.25</td><td>true</td></tr>
<tr><td>loop-amt</td><td>⏱</td><td>0 == i && amt == i_pre && ctr - 1 > i_pre && i_pre <= amt && 0 != i_pre && i_pre <= ctr && amt != amt_pre && ctr - 1 > amt_pre && amt_pre <= amt && 0 != amt_pre && amt_pre <= ctr && ctr - 1 != 1 && 1 != ctr && 1 != amt && 1 == ctr + amt || ... || amt == i && 1 == ctr && 1 != amt && 1 == ctr + amt</td></tr>
<tr><td>loop-disjoint</td><td>0.02</td><td>true</td></tr>
<tr><td>loop-inter</td><td>4.63</td><td>0 == x && 0 != y || 0 == y</td></tr>
<tr><td>loop-simple</td><td>0.06</td><td>true</td></tr>
<tr><td>matrix</td><td>0.71</td><td>0 == y</td></tr>
<tr><td>nested-counter</td><td>6.25</td><td>0 != c && c != t || c == t; c != x && c <= x && 1 != x && t == x || ... ||</td></tr>
</table>

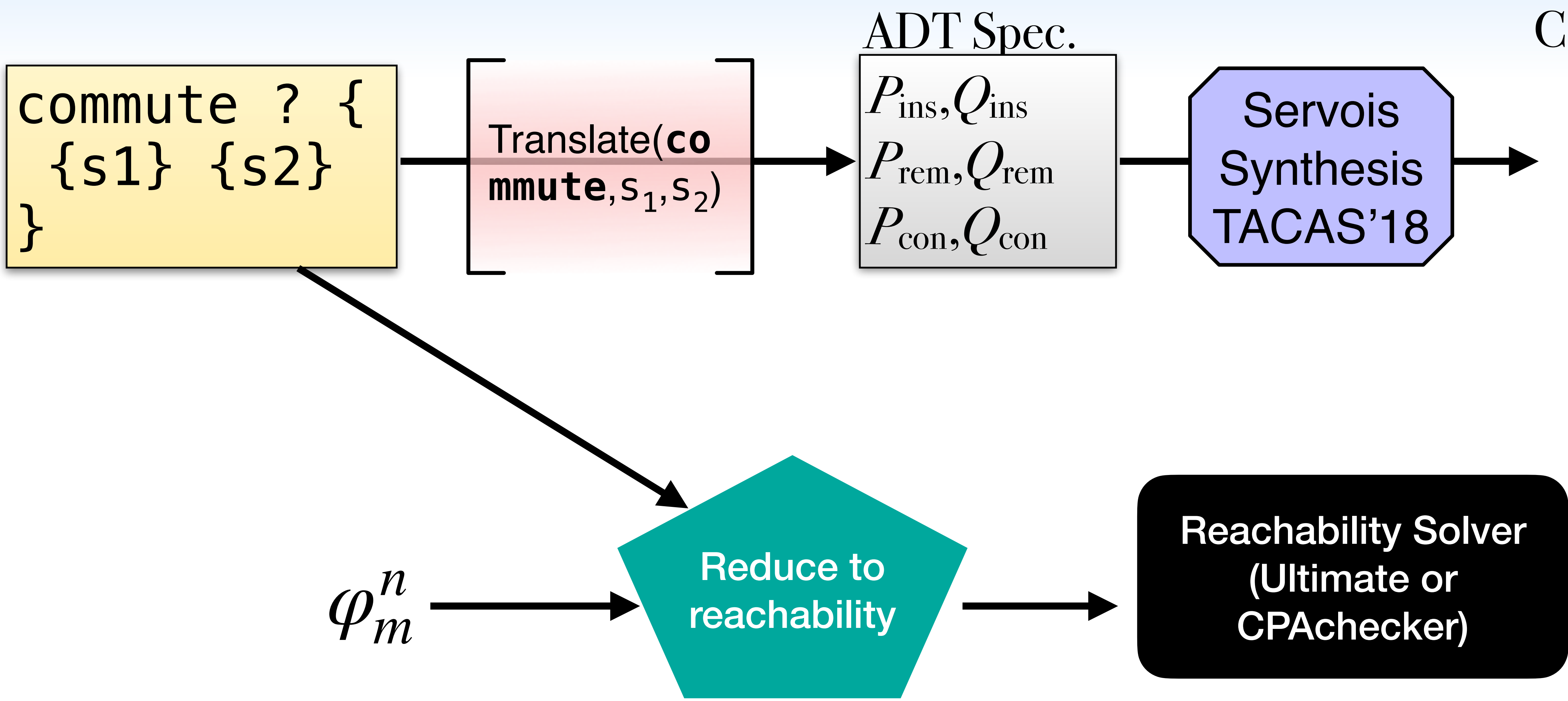


<table>
<tr><td colspan="5">Group 2: Automatically Verified Commute Conditions. Benchmarks for which inference output was suboptimal.</td></tr>
<tr><th>Program</th><th>Time (s)</th><th>Verified?</th><th>Complete?</th><th>Provided Condition</th></tr>
<tr><td>array1</td><td>0.02</td><td>✓</td><td>✓</td><td>r[0] <= 0 || r[0] == 1 && y == 2</td></tr>
<tr><td>calc</td><td>0.07</td><td>✓</td><td>?</td><td>c > 0</td></tr>
<tr><td>counter</td><td>0.02</td><td>✗</td><td>—</td><td>true</td></tr>
<tr><td>even-odd</td><td>0.04</td><td>✓</td><td>✓</td><td>y % 2 == 0</td></tr>
<tr><td>linear-bool</td><td>0.02</td><td>✓</td><td>✓</td><td>y < 0 - 3 * x && x == 2 || y >= 0 && x == 3</td></tr>
<tr><td>linear-cond</td><td>0.02</td><td>✓</td><td>✓</td><td>y > 0 || 0 == y && x + 2 == z</td></tr>
<tr><td>loop-amt</td><td>0.04</td><td>✓</td><td>✗</td><td>i % 2 == 0 && (ctr > 0 && amt > 0 || ctr <= 0 && amt < -ctr)</td></tr>
<tr><td>nested-counter</td><td>0.05</td><td>✓</td><td>✓</td><td>First commute block: 0 != c && c != t || c == t</td></tr>
<tr><td>(cont.)</td><td></td><td>✓</td><td>✗</td><td>Second commute block: x == t && (x > c || x == c && x > 1)</td></tr>
<tr><td>simple</td><td>0.04</td><td>✓</td><td>✗</td><td>c > a</td></tr>
</table>


commute ? { {s1} {s2}
Translate(commute,s1,s2)
$P_{\text{ins}}, Q_{\text{ins}}$
$P_{\text{rm}}, Q_{\text{rm}}$
Servois Synthesis TACAS'18
φ_m^n


commute ? {
{s1} {s2}
}
Translate(commute,s1,s2)
ADT Spec.
$P_{\text{ins}}, Q_{\text{ins}}$
$P_{\text{rem}}, Q_{\text{rem}}$
$P_{\text{con}}, Q_{\text{con}}$
Servois Synthesis TACAS'18
Commute Cond.
φ_m^n
φ_m^n
Reduce to reachability
Reachability Solver (Ultimate or CPAchecker)





Decomposing Data Structure Commutativity Proofs with *mn*-Differencing

Eric Koskinen[1(✉)] and Kshitij Bansal[2]

[1] Stevens Institute of Technology, Hob
eric.koskinen@stevens.edu

VMCAI 2021

Future

Beyond the interpreter.

Combine with promises/futures?

Beyond N-way commute blocks?

Combine with invariant generation:

```
{ y < 0 } /* Example invariant */
commute _ {
  { y = y + 3*x; }  /* if x is negative, this will reduce y */
  { if (y<0) { x=2; } else { x=3; } } /* sensitive to whether y went below 0 */
}
```

Future

Beyond the interpreter.

Combine with promises/futures?

Beyond N-way commute blocks?

Combine with invariant generation:

```
{ y < 0 } /* Example invariant */
commute _ {
  { y = y + 3*x; }  /* if x is negative, this will reduce y */
  { if (y<0) { x=2; } else { x=3; } } /* sensitive to whether y went below 0 */
}
```

```
0 > y + 3*x && 2 == x
```

Interested in Commutativity?

Workshop at PLDI next month!

PLDI San Diego 2022

Mon 13 - Fri 17 June 2022
San Diego, California, United States

Attending ▾ Program ▾ Tracks ▾ Organization ▾ Sign in Sign up

PLDI 2022 (series) / Commute (series) /

Commutativity Reasoning and Applications

Commute

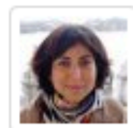
This is the first instance of the Commutativity Reasoning & Applications workshop (Commute 2022).

Commutativity Reasoning is becoming increasingly common and appears in many contexts. Commutativity is used in the design of systems, in the design of data structures, in proof methodologies, in parallel execution schemes, etc. The goal of this workshop is to bring together researchers that are working in a variety of areas, with a common need for commutativity, to share ideas and goals. We aim to include researchers who work on commutativity in many contexts: compilers, program logics, automata, concurrency, distributed systems/CRDTs, ML applications, etc.

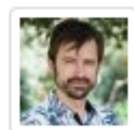
The workshop will be held on Monday June 13th and Tuesday June 14th.

Organizing Committee

Constantin Enea
Ecole Polytechnique / LIX / CNRS
France



Azadeh Farzan
University of Toronto
Canada



Eric Koskinen
Stevens Institute of Technology

Call for Papers

The workshop is open to all who are interested and/or working in the area of Commutativity Reasoning and Applications. This includes researchers in

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Interested in Commutativity?

Come visit!

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

CYPRESS

CYBERSECURITY
PROGRAMMING LANGUAGES
AND SYSTEMS AT STEVENS

Other recent things I didn't have time for …

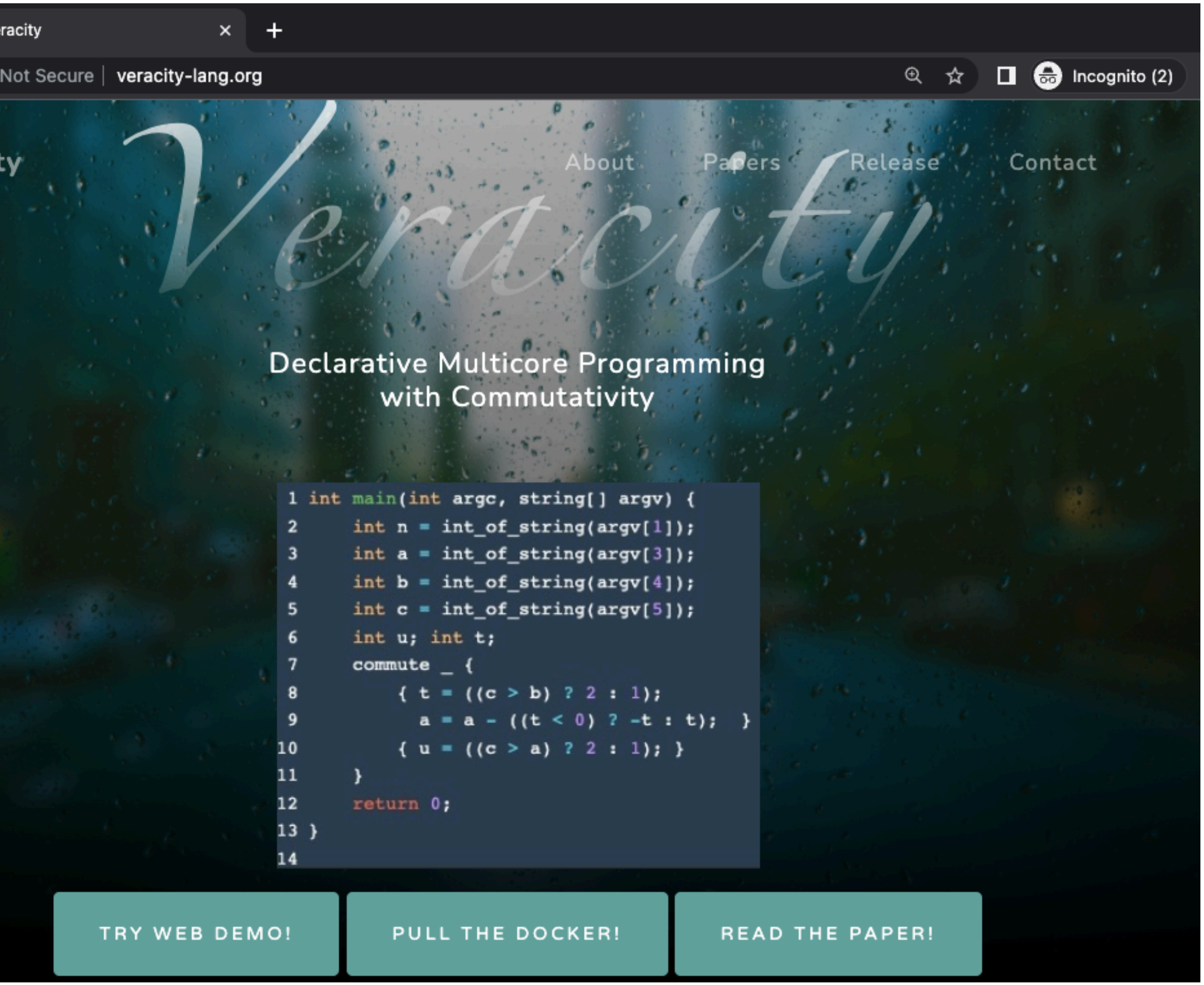
- Automatic **temporal** verification — *CAV'11, POPL'11, PLDI'13, LICS'14, LICS'18*
- Automatic **relational** verification — *PLDI'17, OOPSLA'19, CAV'21*
- Automatic crash recoverability — *POPL'16*
- Verifying binaries — *APLAS'21, IEEE S&P'21*
- Transactional implementations — *APLAS'19, PODC'17, VMCAI'17, PPoPP'08*
- Semantics of transactions — *POPL'10, PLDI'15*

www.erickoskinen.com

Thank you!

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Thank you!



www.veracity-lang.org

www.erickoskinen.com