Concurrent Data Structures: Semantics and Relaxations

Ana Sokolova

UNIVERSITY of SALZBURG

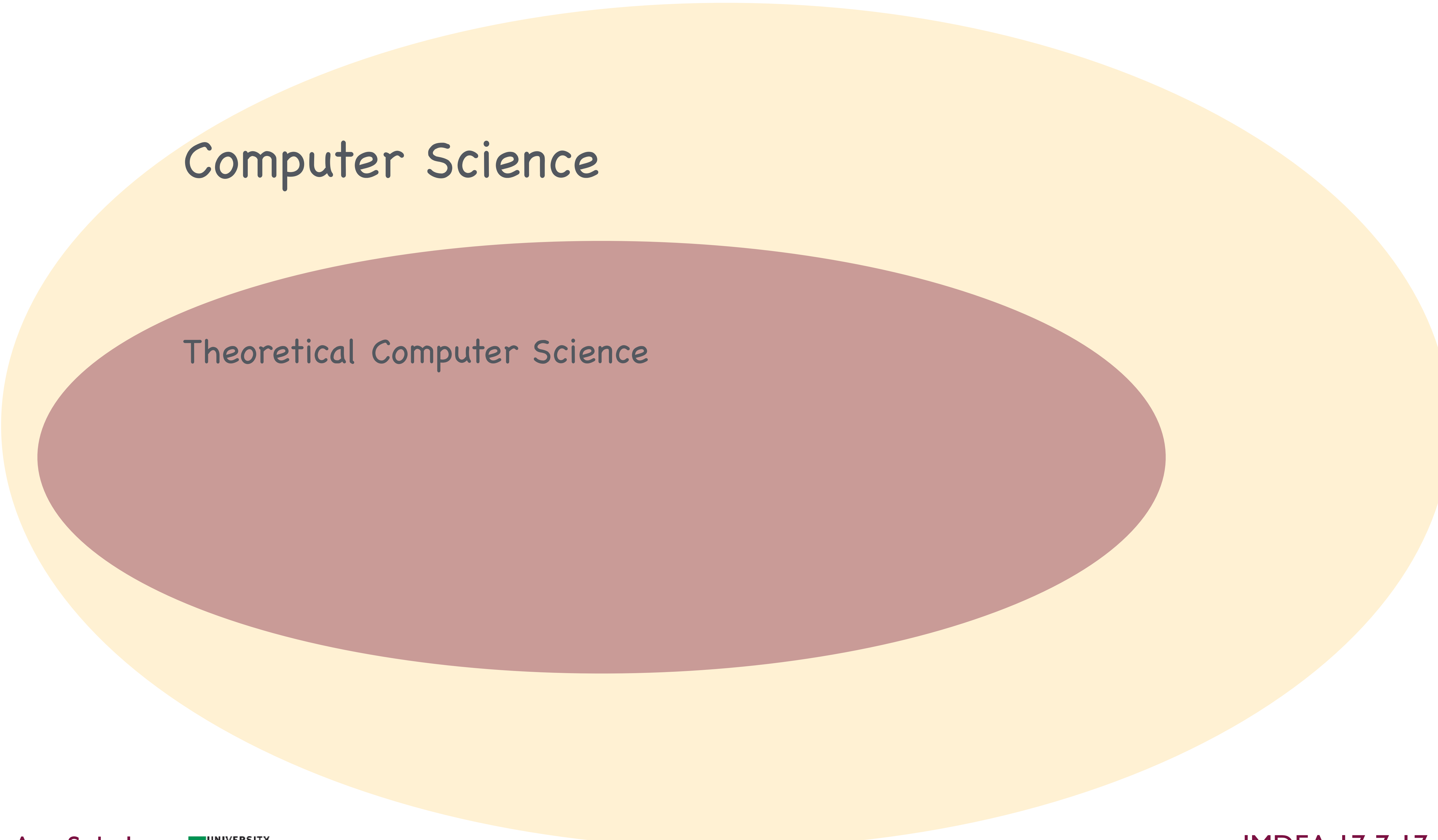
IMDEA, Madrid, 17.7.17

Background big picture

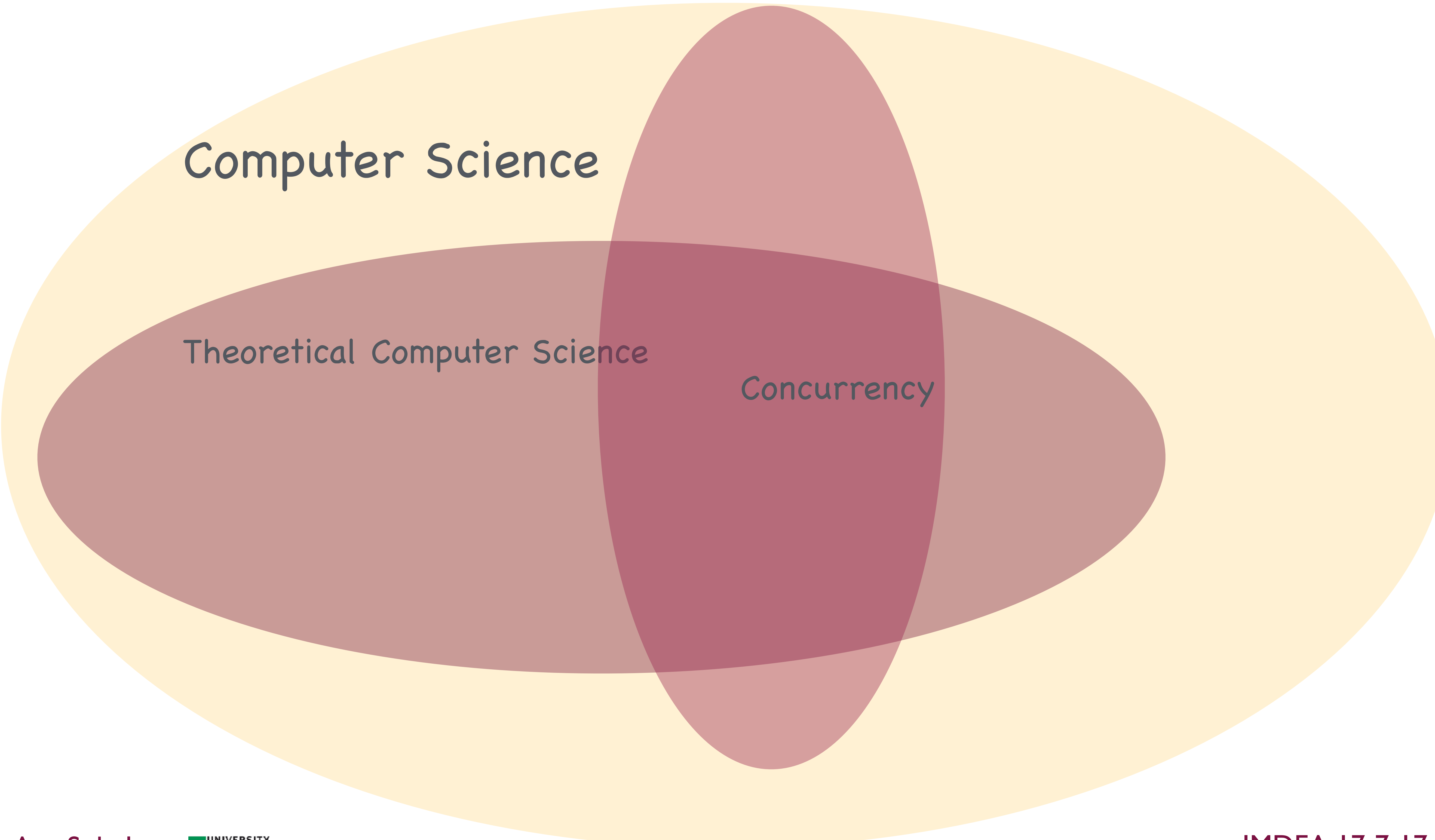
Background big picture

Computer Science

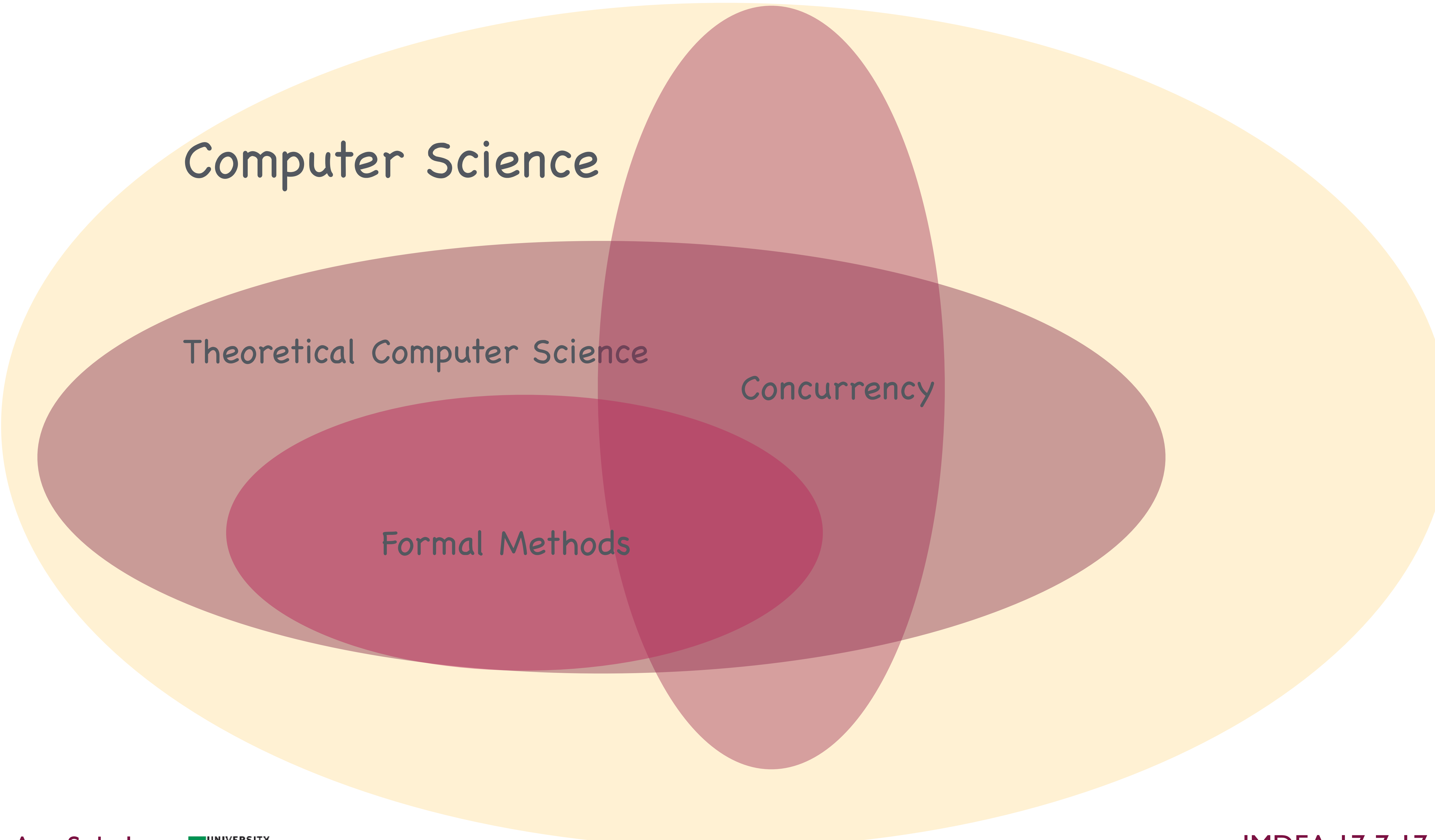
Background big picture



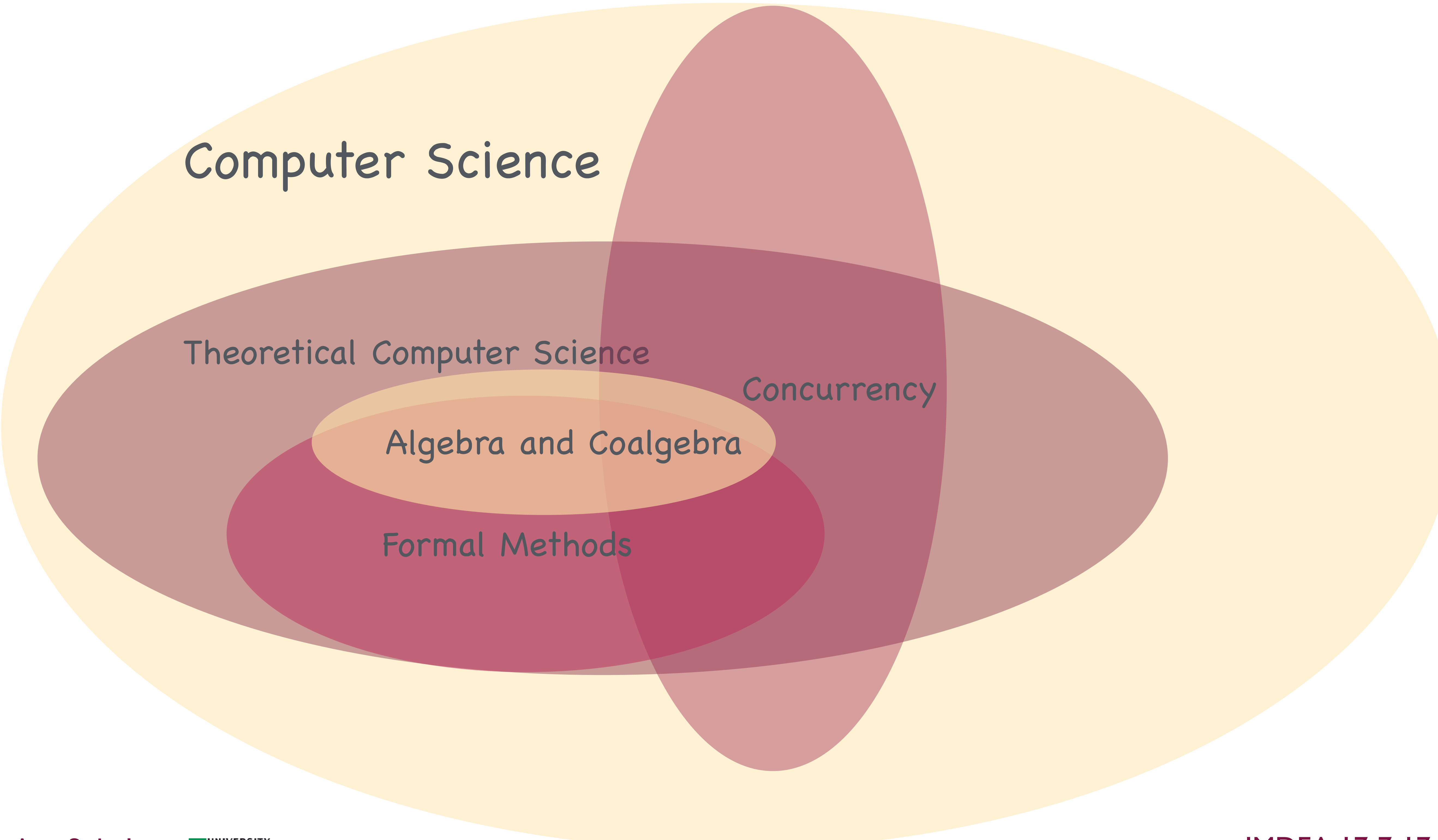
Background big picture



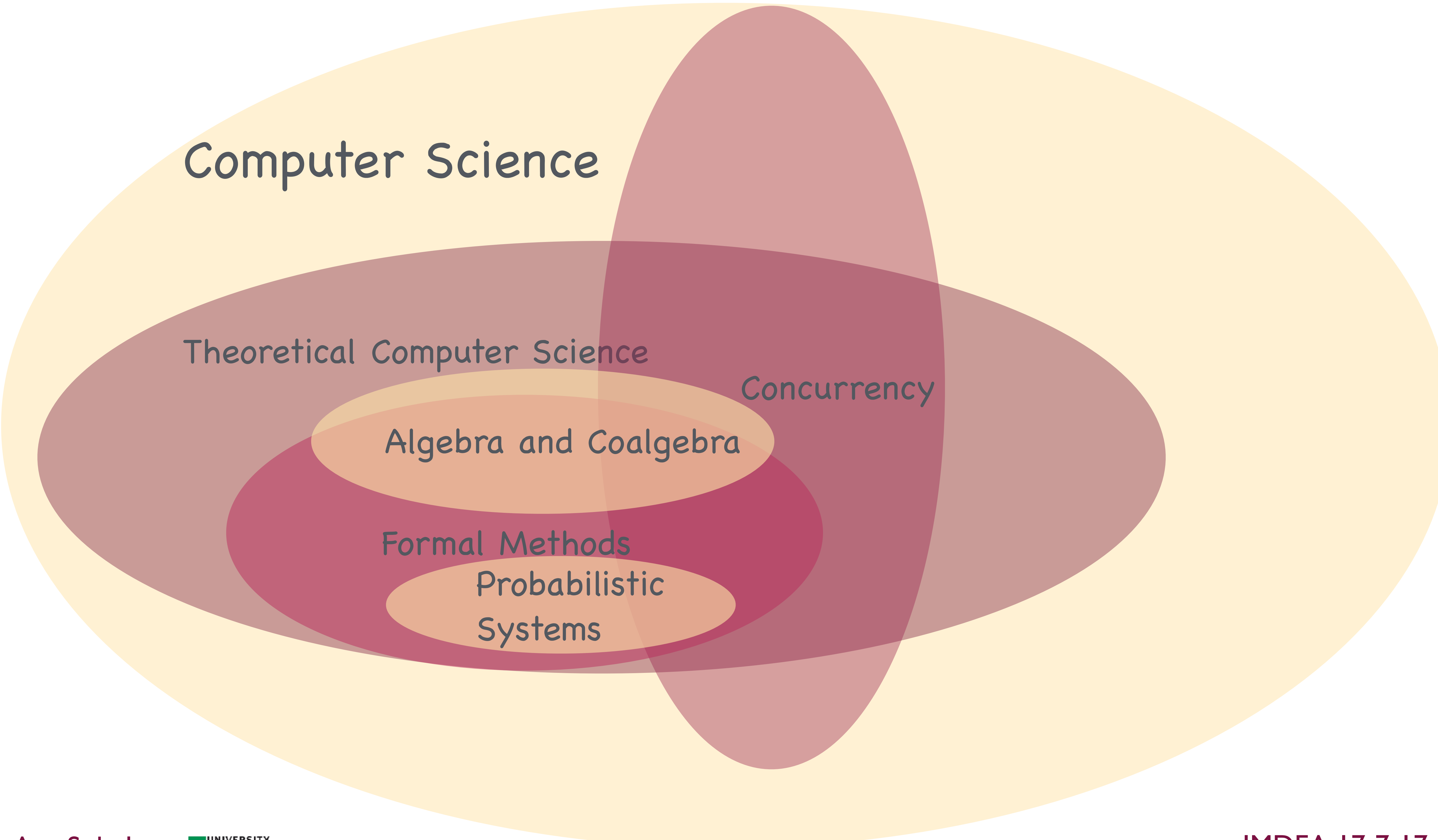
Background big picture



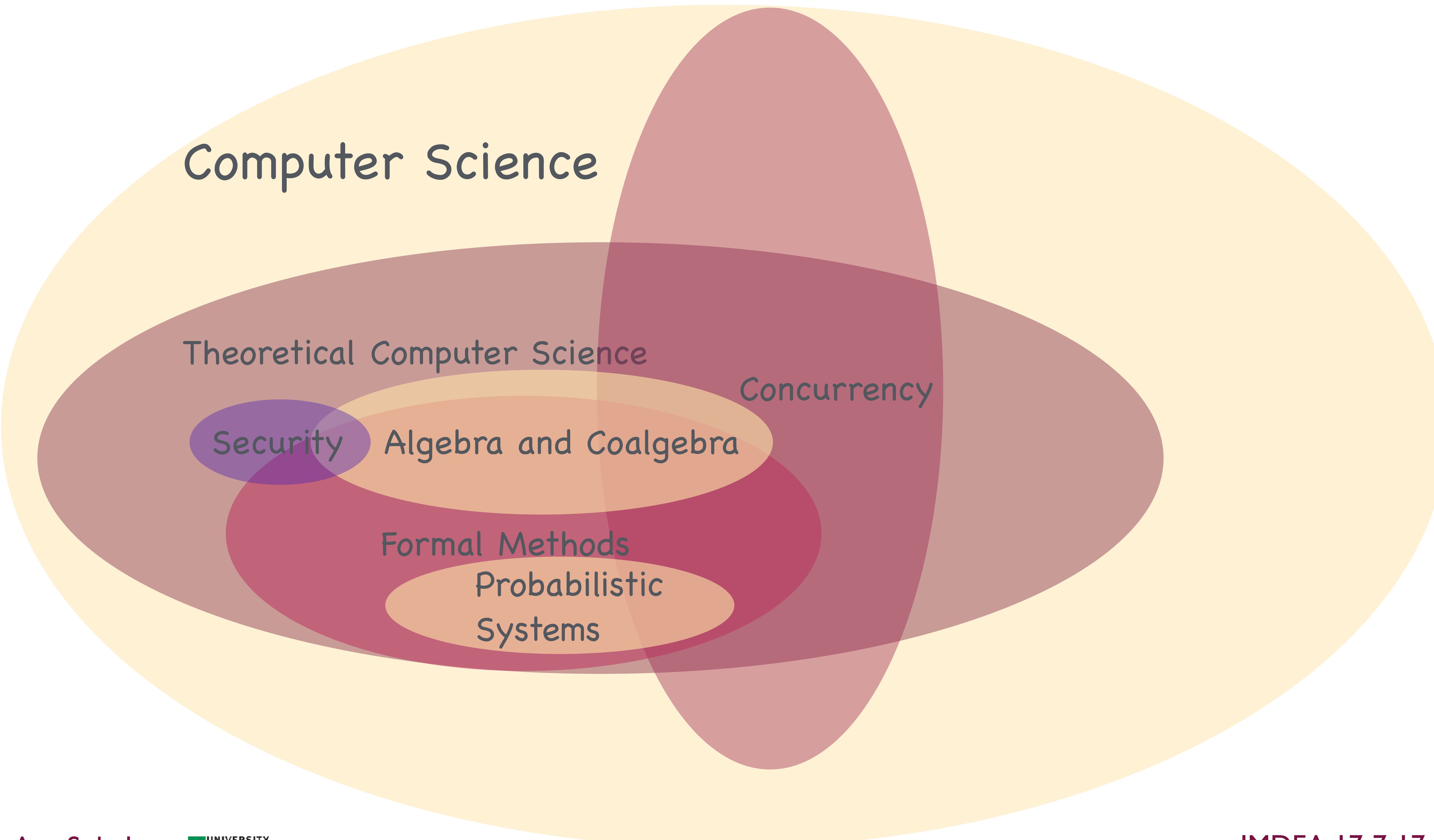
Background big picture



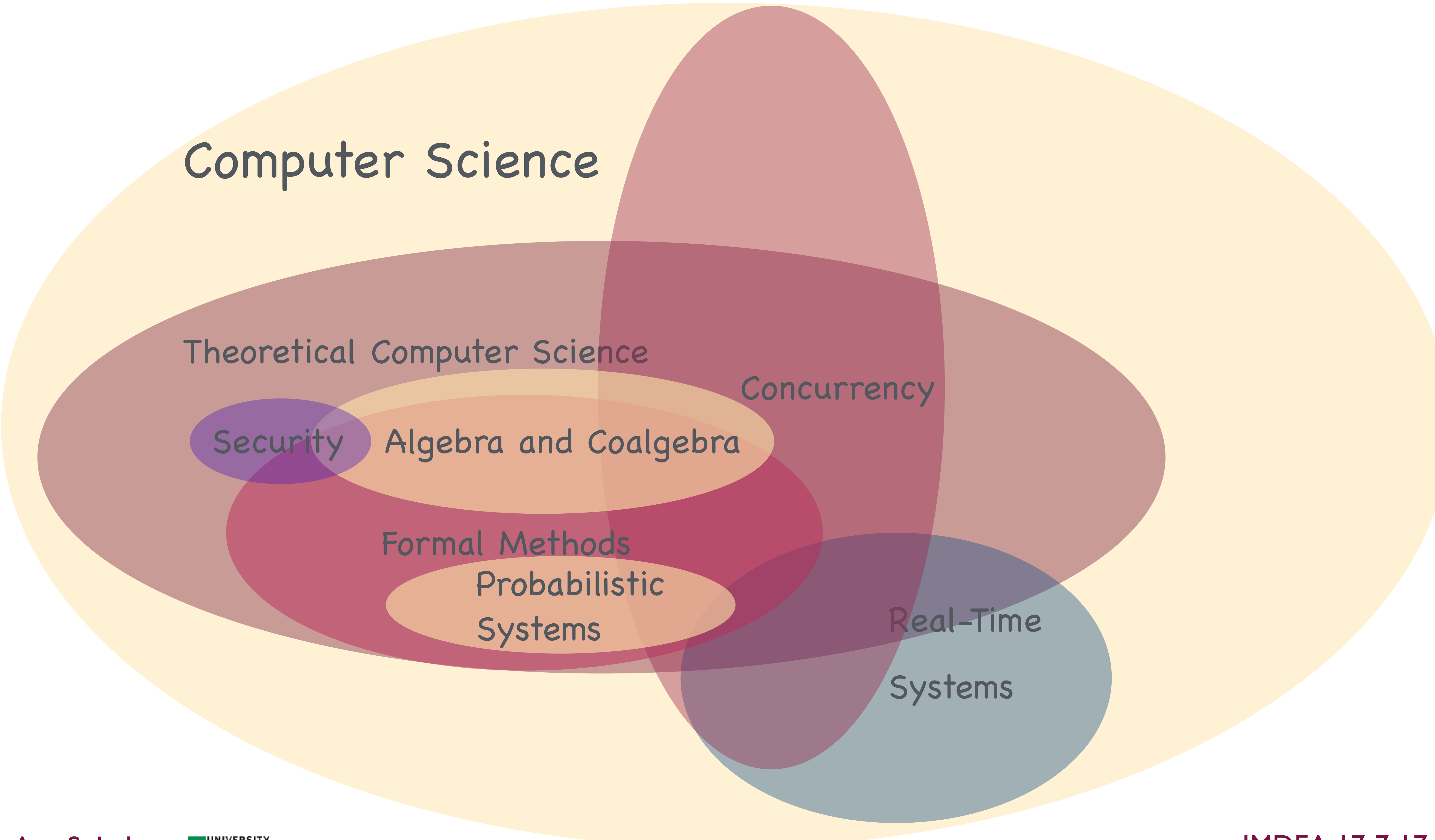
Background big picture



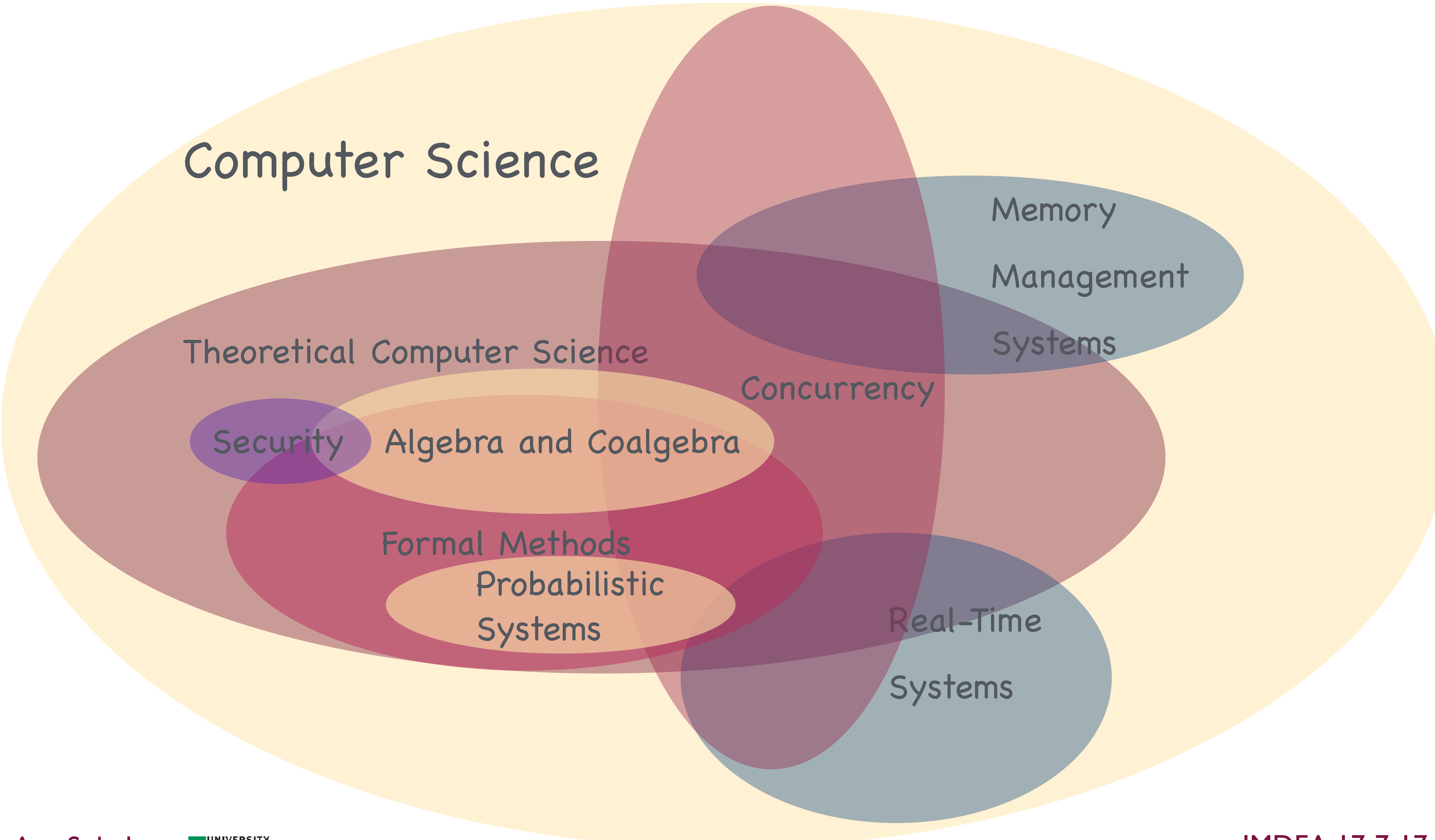
Background big picture



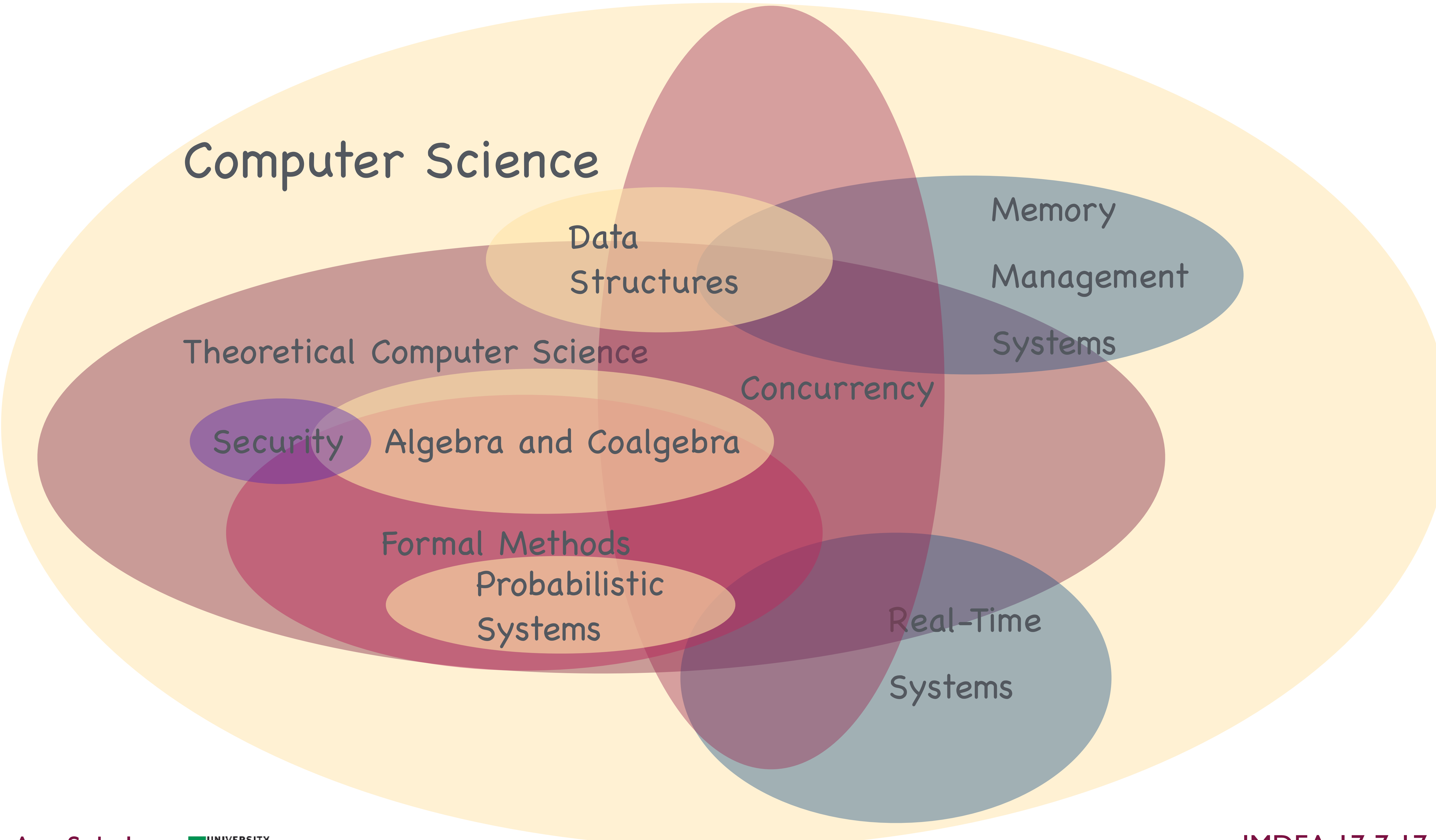
Background big picture



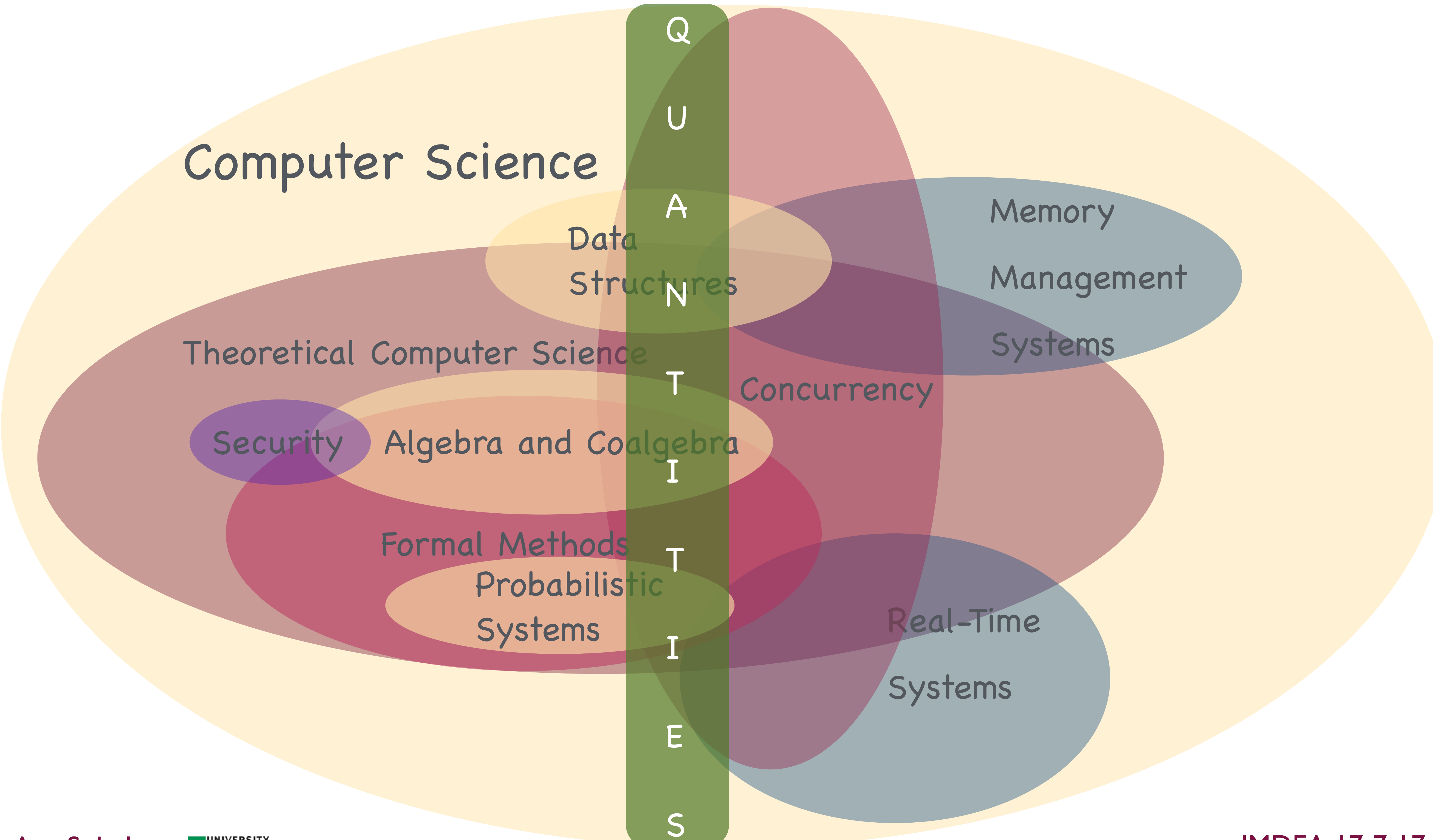
Background big picture



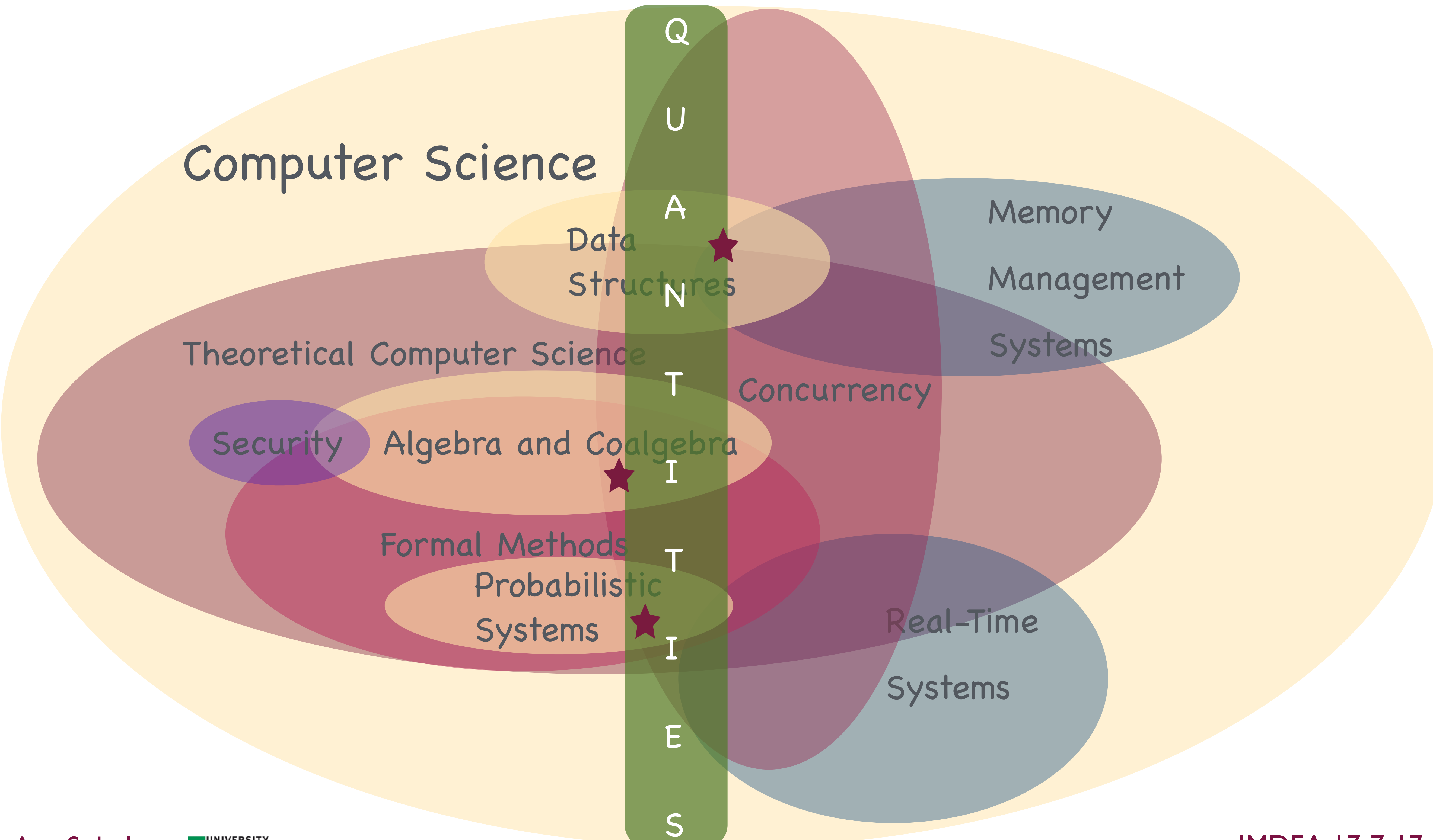
Background big picture



Background big picture



Favourites



Q
U
A
N
T
I
T
I
E
S
Computer Science
Data
Structures
Memory
Management
Systems
Theoretical Computer Science
Concurrency
Security
Algebra and Coalgebra
Formal Methods
Probabilistic
Systems
Real-Time
Systems

Concurrent Data Structures: Semantics and Relaxations

Ana Sokolova UNIVERSITY of SALZBURG

IMDEA, Madrid, 17.7.17

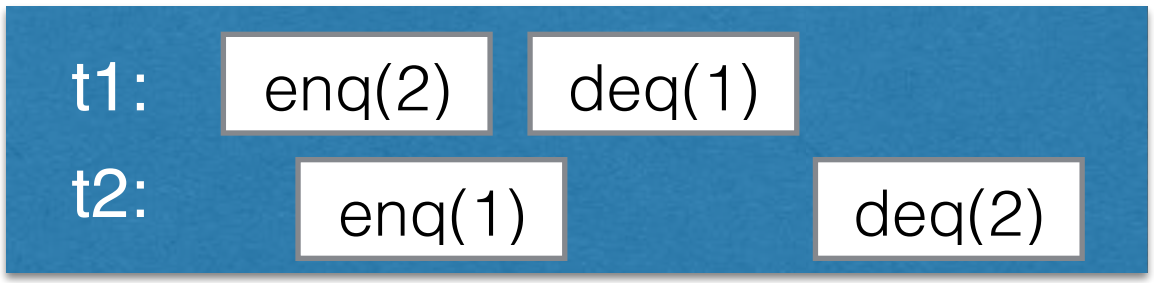
Concurrent Data Structures: Correctness and Performance

Semantics of concurrent data structures

Semantics of concurrent data structures

e.g. pools, queues, stacks

Semantics of concurrent data structures



e.g. pools, queues, stacks

Semantics of concurrent data structures

t1: enq(2) deq(1)

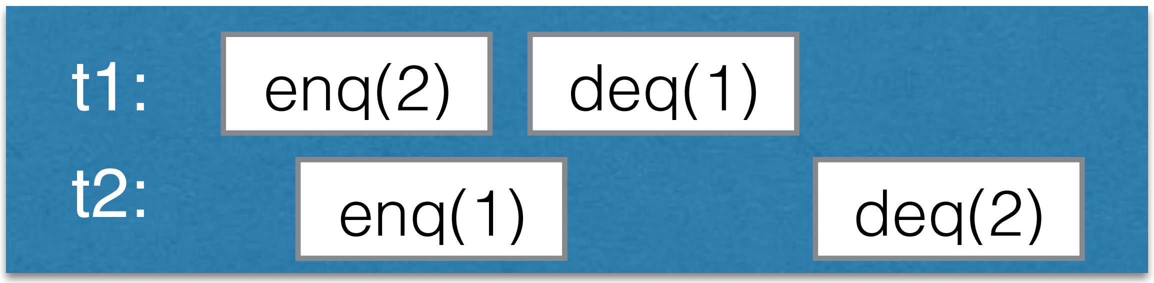
t2: enq(1) deq(2)

e.g. pools, queues, stacks

- Sequential specification = set of legal sequences

- Consistency condition = e.g. linearizability / sequential consistency

Semantics of concurrent data structures



e.g. pools, queues, stacks

- Sequential specification = set of legal sequences

e.g. queue legal sequence
enq(1)enq(2)deq(1)deq(2)

- Consistency condition = e.g. linearizability / sequential consistency

Semantics of concurrent data structures

t1: enq(2) deq(1)

t2: enq(1) deq(2)

e.g. pools, queues, stacks

- Sequential specification = set of legal sequences

e.g. queue legal sequence
enq(1)enq(2)deq(1)deq(2)

- Consistency condition = e.g. linearizability / sequential consistency

e.g. the concurrent history above is a linearizable queue concurrent history

Consistency conditions

Linearizability [Herlihy,Wing '90]

↓

Sequential Consistency [Lamport'79]

Consistency conditions

there exists a legal sequence that preserves precedence

Linearizability [Herlihy,Wing '90]

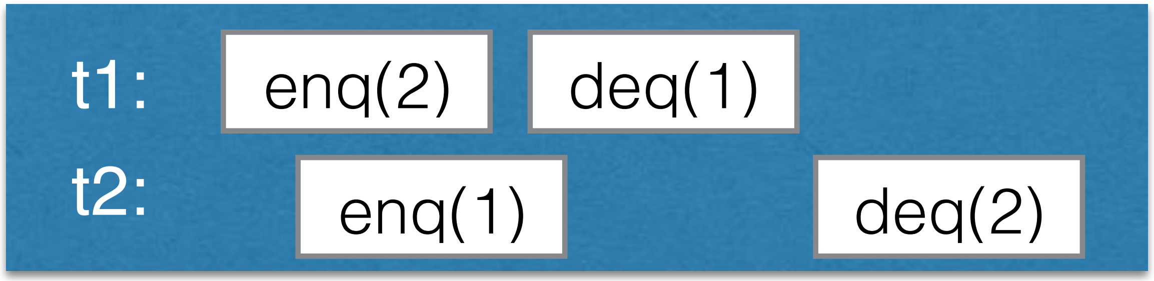
↓

Sequential Consistency [Lamport'79]

Consistency conditions

there exists a legal sequence that preserves precedence

Linearizability [Herlihy,Wing '90]

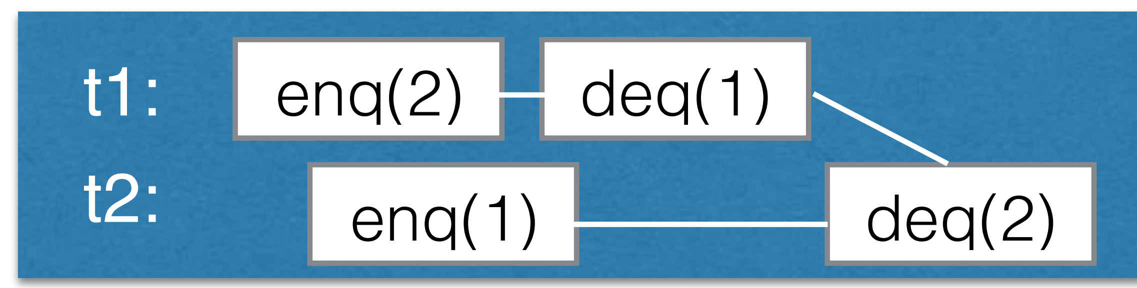


Sequential Consistency [Lamport'79]

Consistency conditions

there exists a legal sequence that preserves precedence

Linearizability [Herlihy,Wing '90]

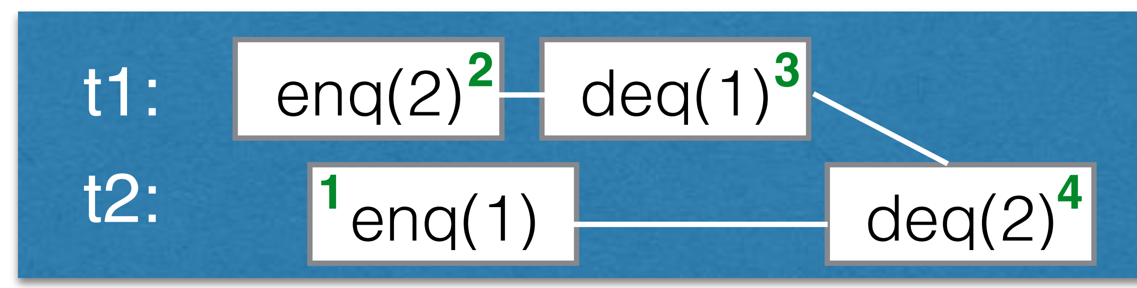


Sequential Consistency [Lamport'79]

Consistency conditions

there exists a legal sequence that preserves precedence

Linearizability [Herlihy,Wing '90]



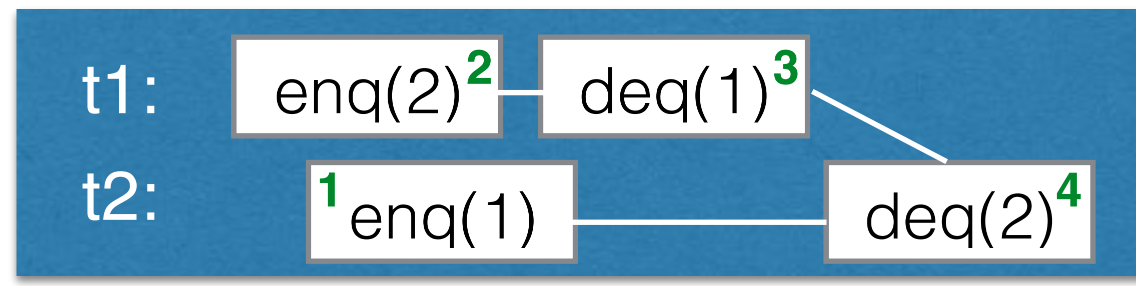
↓

Sequential Consistency [Lamport'79]

Consistency conditions

there exists a legal sequence that preserves precedence

Linearizability [Herlihy,Wing '90]



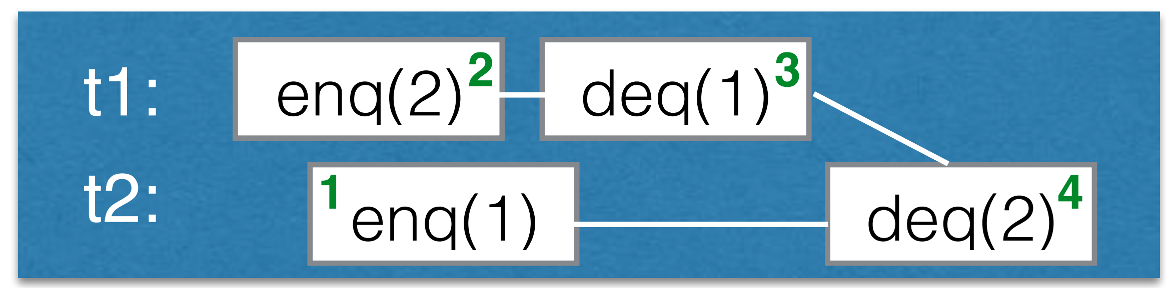
Sequential Consistency [Lamport'79]

there exists a legal sequence that preserves per-thread precedence (program order)

Consistency conditions

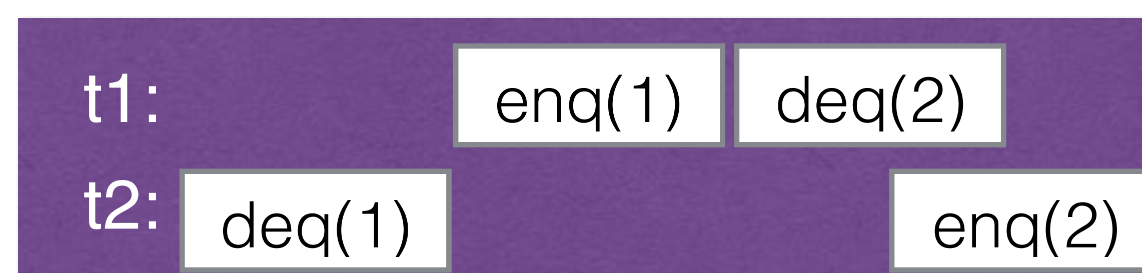
there exists a legal sequence that preserves precedence

Linearizability [Herlihy,Wing '90]



Sequential Consistency [Lamport'79]

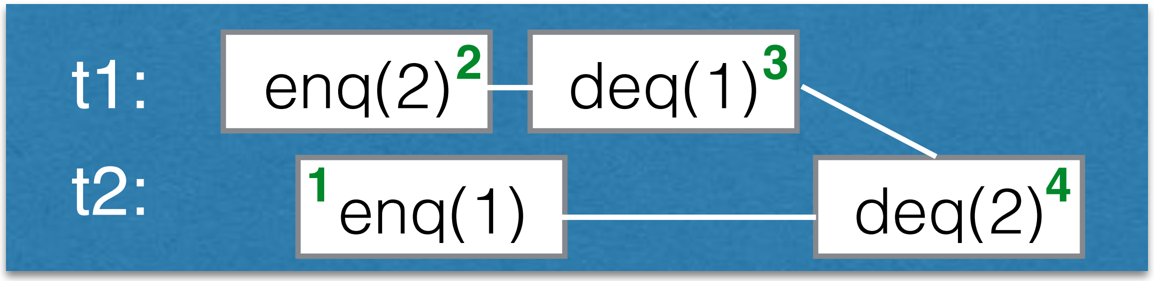
there exists a legal sequence that preserves per-thread precedence (program order)



Consistency conditions

there exists a legal sequence that preserves precedence

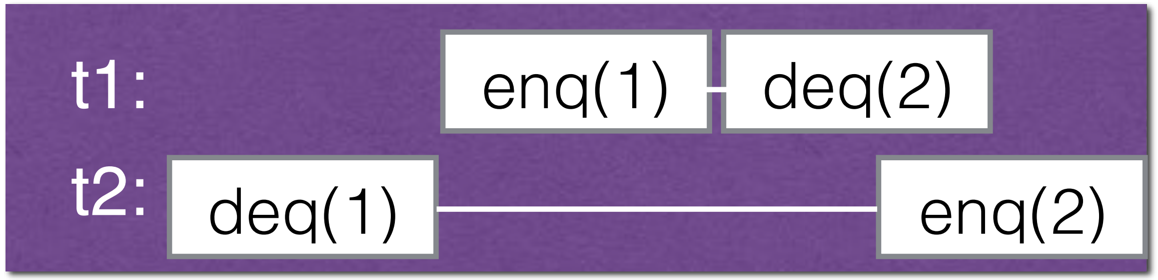
Linearizability [Herlihy,Wing '90]



↓

Sequential Consistency [Lamport'79]

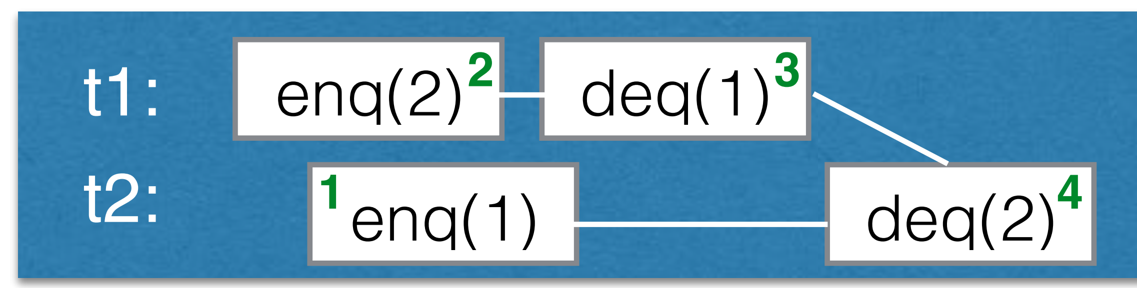
there exists a legal sequence that preserves per-thread precedence (program order)



Consistency conditions

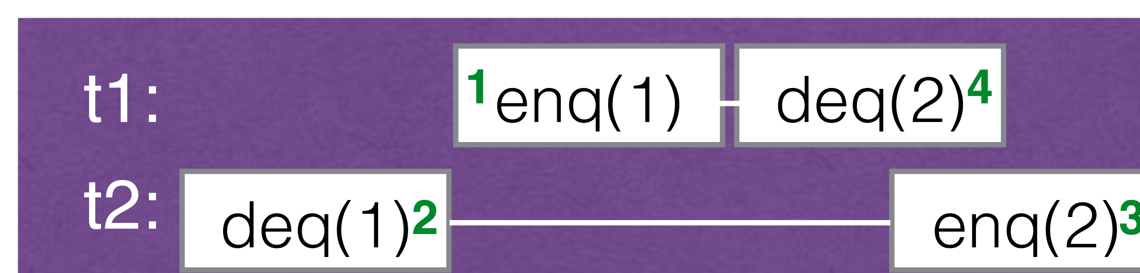
there exists a legal sequence that preserves precedence

Linearizability [Herlihy,Wing '90]

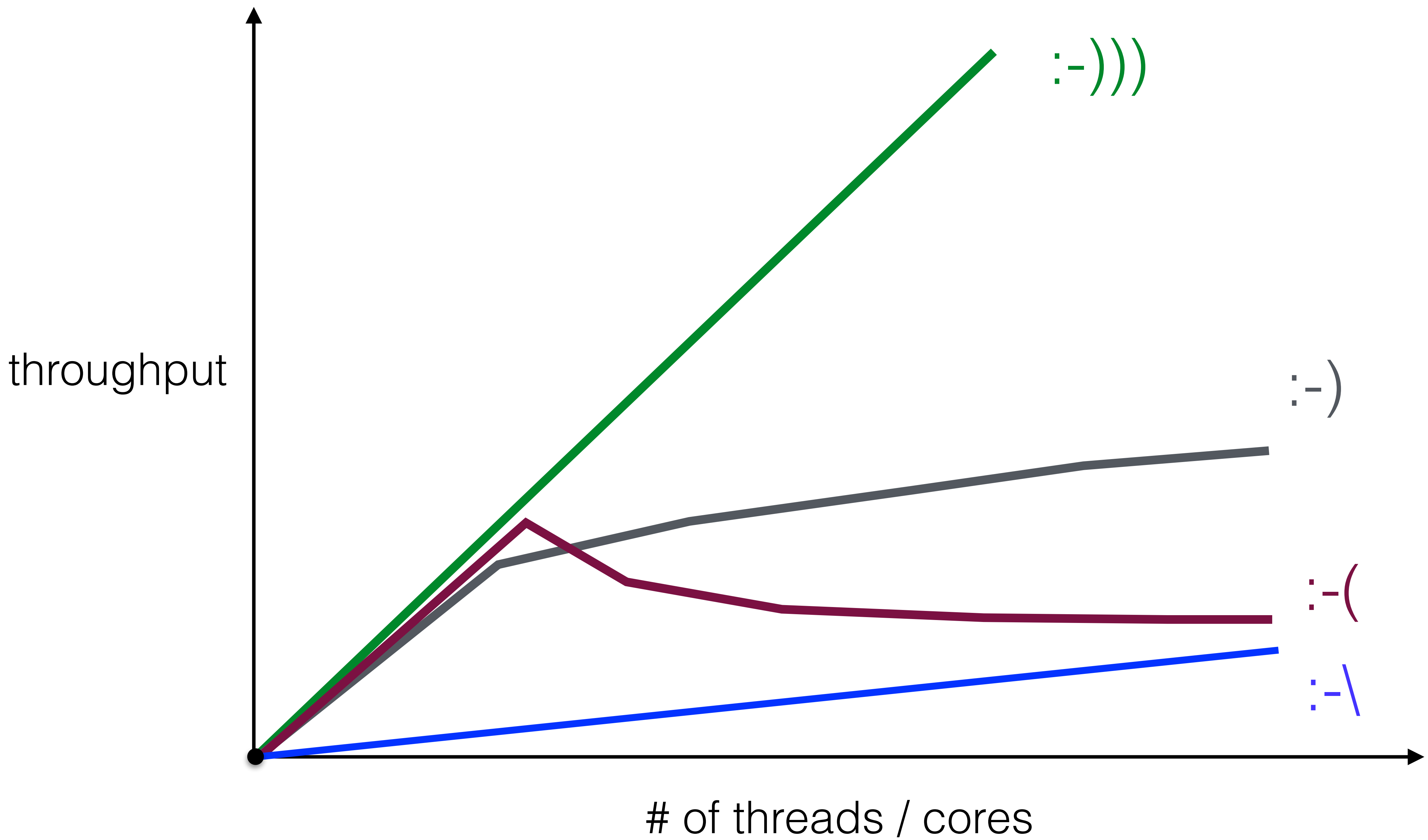


Sequential Consistency [Lamport'79]

there exists a legal sequence that preserves per-thread precedence (program order)



Performance and scalability



Relaxations allow trading

correctness
for
performance

Relaxations allow trading

correctness
for
performance

provide the potential for better-performing implementations

Relaxing the semantics

Relaxing the semantics

- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

Relaxing the semantics

Quantitative relaxations
POPL13

- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

Relaxing the semantics

not
“sequentially
correct”

Quantitative relaxations
POPL13

- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

Relaxing the semantics

not "sequentially correct"

Quantitative relaxations
POPL13

- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

Local linearizability
CONCUR16

Relaxing the semantics

not "sequentially correct"

Quantitative relaxations
POPL13

- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

Local linearizability
CONCUR16

too weak

Relaxing the semantics

not "sequentially correct"

Quantitative relaxations
POPL13

- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

for queues/stacks only
(feel free to ask for more)

too weak

Local linearizability
CONCUR16

Relaxing the sequential specification

Relaxing the sequential specification

Quantitative relaxations (POPL13)

Goal

- trade correctness for performance
- in a controlled way with quantitative bounds

measure the
error from correct
behaviour

Goal

Stack - incorrect behavior

push(a)push(b)push(c)pop(a)pop(b)

- trade correctness for performance
- in a controlled way with quantitative bounds

measure the error from correct behaviour

Goal

Stack - incorrect behavior

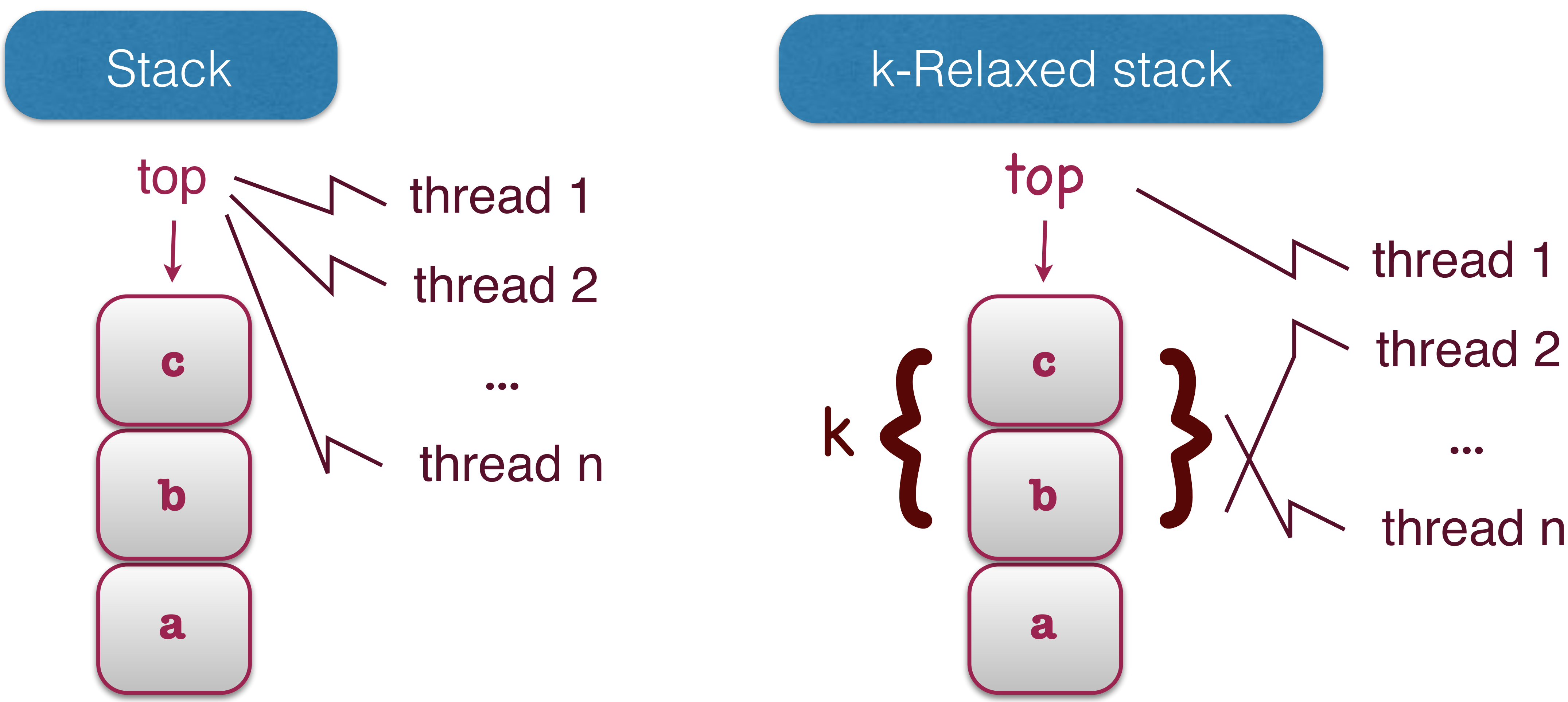
push(a)push(b)push(c)pop(a)pop(b)

- trade correctness for performance
- in a controlled way with quantitative bounds

measure the error from correct behaviour

correct in a relaxed stack
... 2-relaxed? 3-relaxed?

How can relaxing help?



What we have

- Framework
- Generic examples
- Concrete relaxation examples
- Efficient concurrent implementations

for semantic relaxations

out-of-order / stuttering

stacks, queues, priority queues,.. / CAS, shared counter

of relaxation instances

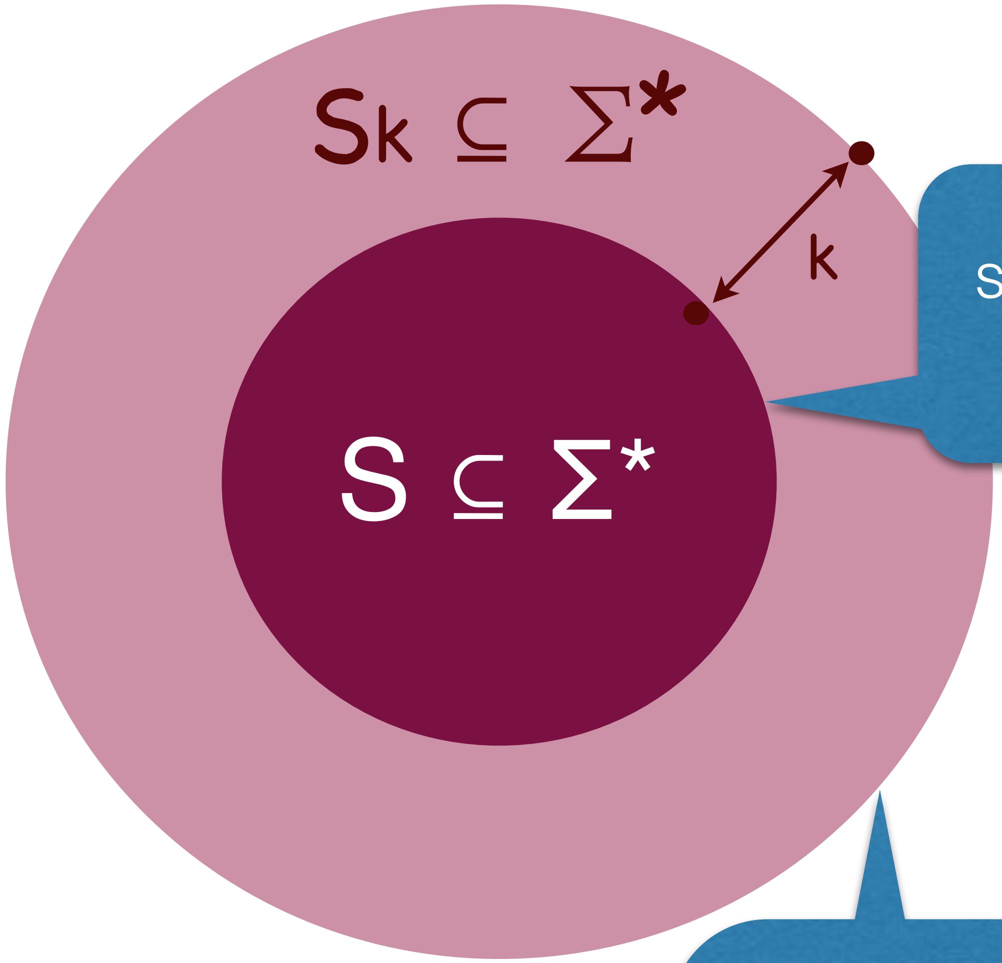
The big picture

$S \subseteq \Sigma^*$

sequential specification
legal sequences

Σ - methods with arguments

The big picture



sequential specification
legal sequences

relaxed sequential specification
sequences at distance up to k from S

Σ - methods with arguments

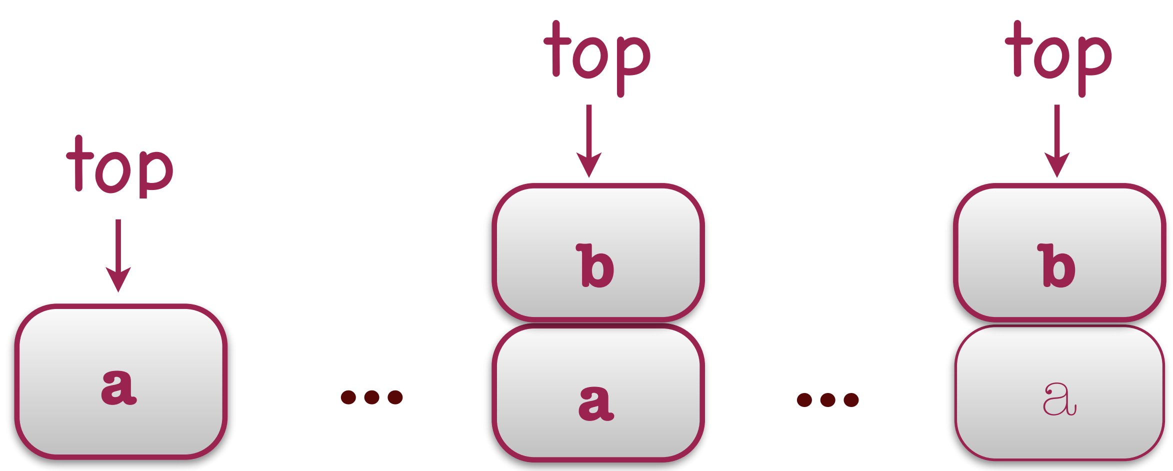
Syntactic distances do not help

push(a)[push(i)pop(i)]^{n}push(b)[push(j)pop(j)]^{m}pop(a)

Syntactic distances do not help

push(a)[push(i)pop(i)]^{n}push(b)[push(j)pop(j)]^{m}pop(a)

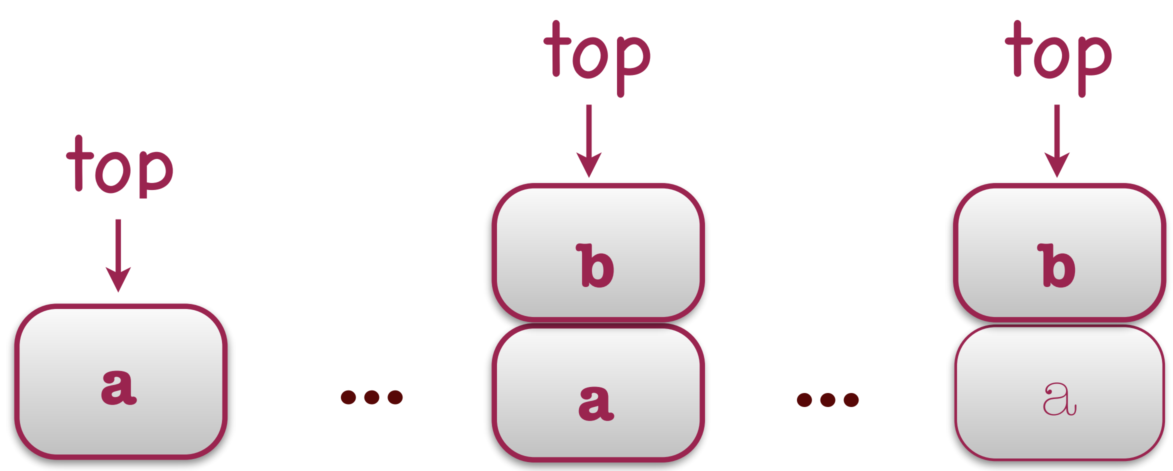
is a 1-out-of-order stack sequence



Syntactic distances do not help

$$\mathrm{push}(a)[\mathrm{push}(i)\mathrm{pop}(i)]^n\mathrm{push}(b)[\mathrm{push}(j)\mathrm{pop}(j)]^m\mathrm{pop}(a)$$

is a 1-out-of-order stack sequence



its permutation distance is min(2n,2m)

Semantic distances need a notion of state

- States are equivalence classes of sequences in S

- Two sequences in S are equivalent iff they have an indistinguishable future

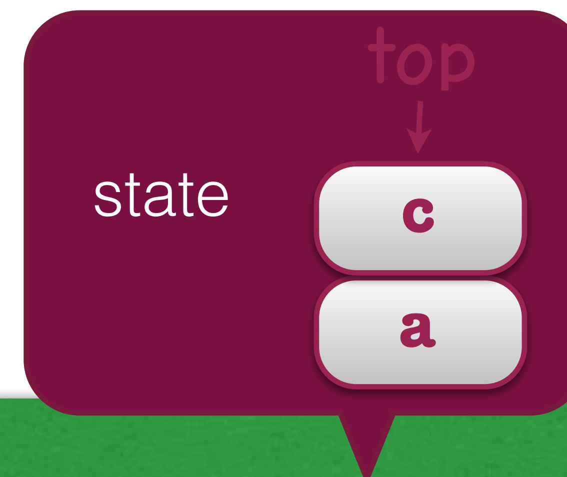
Semantic distances need a notion of state

- States are equivalence classes of sequences in S

- Two sequences in S are equivalent iff they have an indistinguishable future

$$x \equiv y \quad \Leftrightarrow \quad \forall u \in \Sigma^*. \, (xu \in S \; \Leftrightarrow \; yu \in S)$$

Semantic distances need a notion of state



- States are equivalence classes of sequences in S

example: for stack

$$\mathrm{push(a)push(b)pop(b)push(c)} \equiv \mathrm{push(a)push(c)}$$

- Two sequences in S are equivalent iff they have an indistinguishable future

$$x \equiv y \quad \Leftrightarrow \quad \forall u \in \Sigma^*.\,(xu \in S \;\Leftrightarrow\; yu \in S)$$

Semantics goes operational

$S \subseteq \Sigma^*$ is the sequential specification

states

labels

initial state

$LTS(S) = (S/_{\equiv}, \Sigma, \rightarrow, [\varepsilon]_{\equiv})$ with

transition relation

$$[s]_{\equiv} \xrightarrow{m} [sm]_{\equiv} \quad \Leftrightarrow \quad sm \in S$$

Semantics goes operational

$S \subseteq \Sigma^*$ is the sequential specification

states

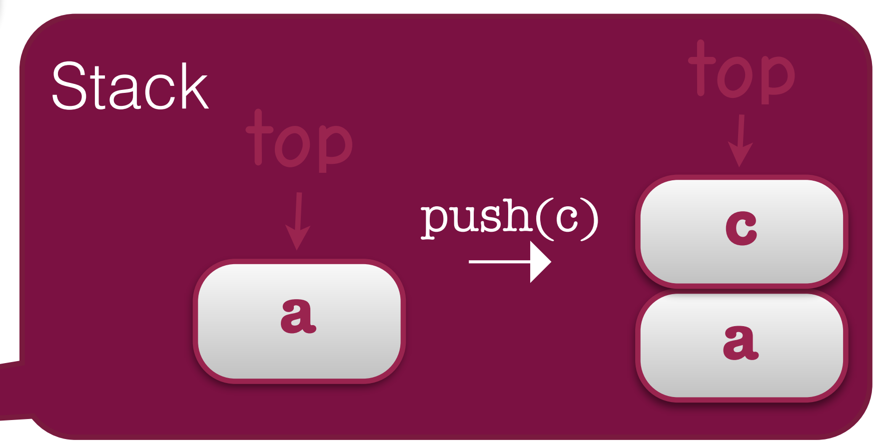
labels

initial state

$\mathrm{LTS}(S) = (S/_\equiv, \Sigma, \rightarrow, [\varepsilon]_\equiv)$ with

transition relation

$$[s]_\equiv \xrightarrow{m} [sm]_\equiv \quad \Leftrightarrow \quad sm \in S$$

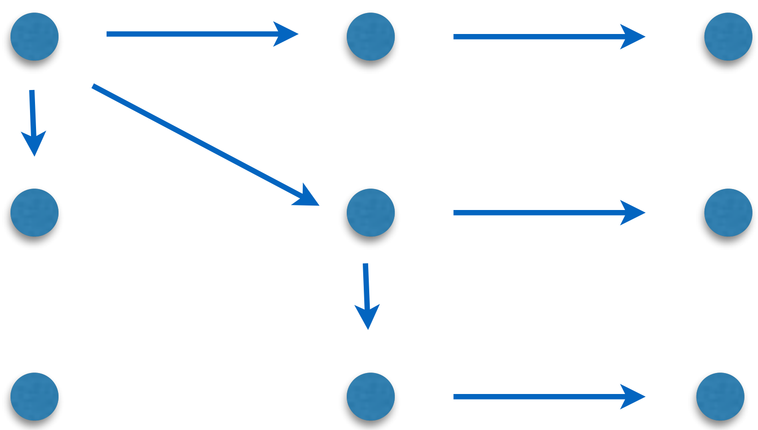


The relaxation framework

- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function

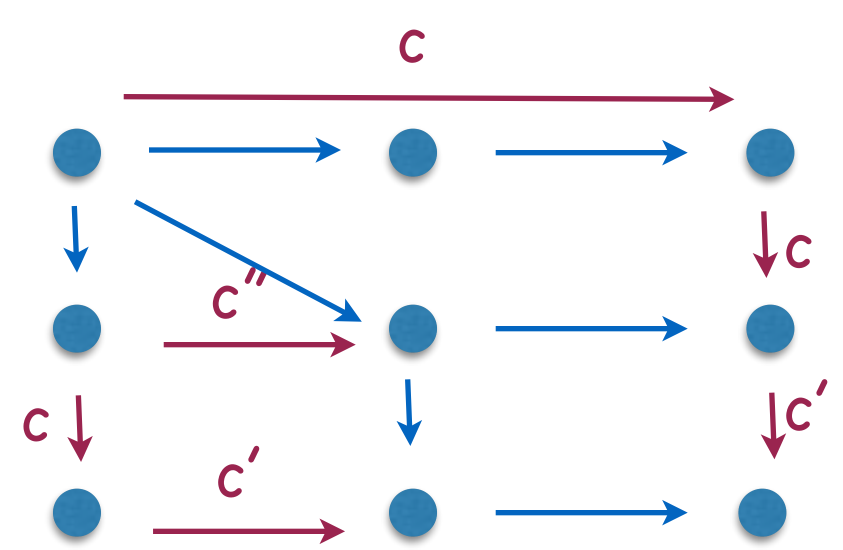
The relaxation framework

- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function



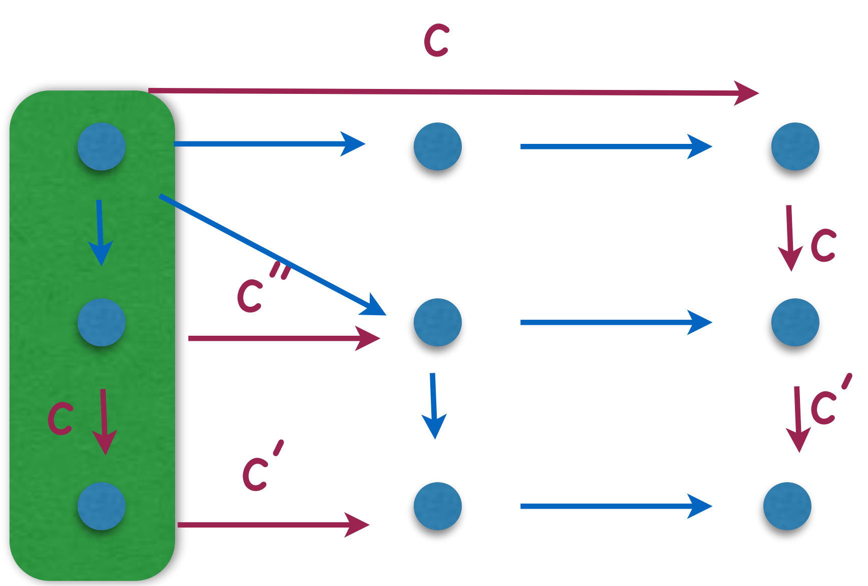
The relaxation framework

- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function



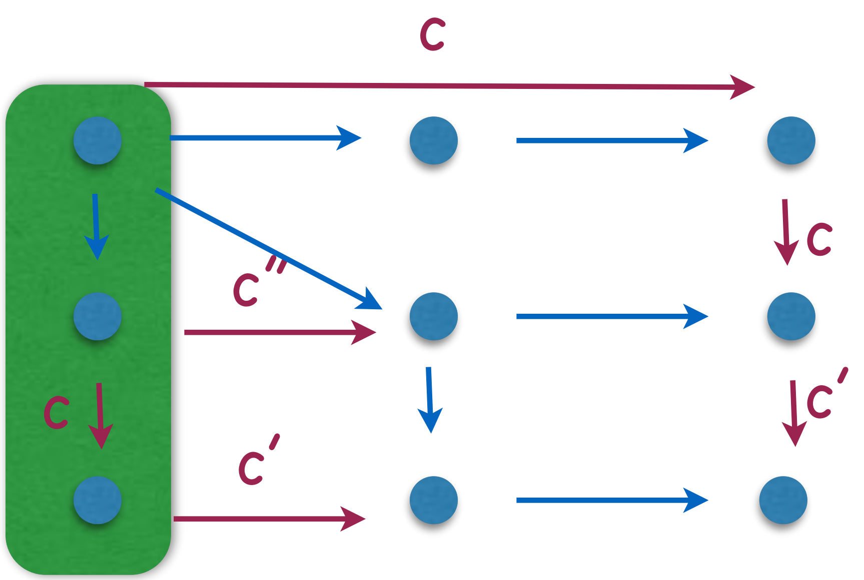
The relaxation framework

- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function



The relaxation framework

- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function



distance - minimal cost on all paths labelled by the sequence

Generic out-of-order

$\texttt{segment\_cost}(\, q \xrightarrow{m} q' \,) = |\mathbf{v}|$

transition cost

Where **v** is a sequence of minimal length s.t.

removing **v** enables a transition

or

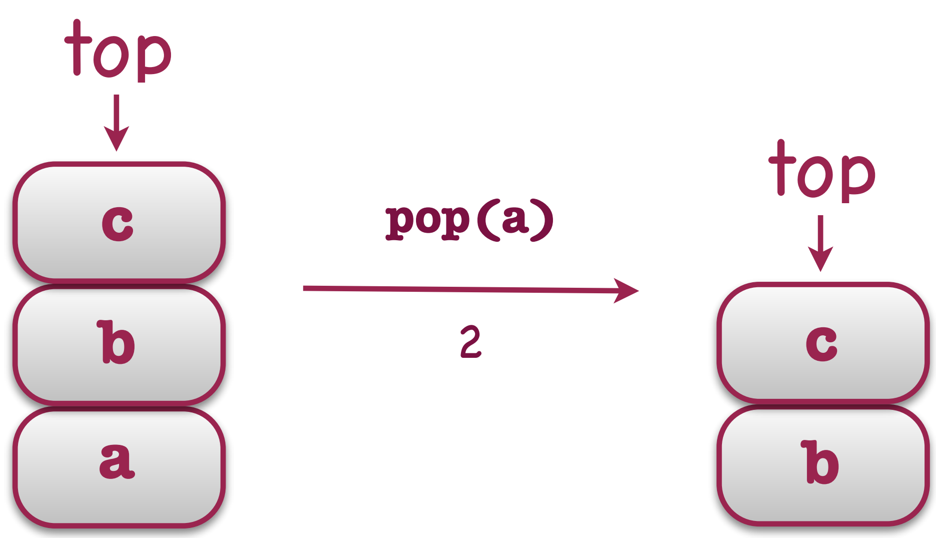
inserting **v** enables a transition

goes with different path costs

Out-of-order stack

Sequence of **push**'s with no matching **pop**

- Canonical representative of a state
- Add incorrect transitions with segment-costs



- Possible path cost functions max, sum,...

also more advanced

Relaxing the Consistency Condition

Relaxing the Consistency Condition

Local Linearizability (CONCUR16)

Local Linearizability main idea

Local Linearizability main idea

- Partition a history into a set of local histories
- Require linearizability per local history

Local Linearizability main idea

Already present in some shared-memory consistency conditions (not in our form of choice)

- Partition a history into a set of local histories
- Require linearizability per local history

Local Linearizability main idea

Already present in some shared-memory consistency conditions
(not in our form of choice)

- Partition a history into a set of local histories
- Require linearizability per local history

Local sequential consistency… is also possible

Local Linearizability main idea

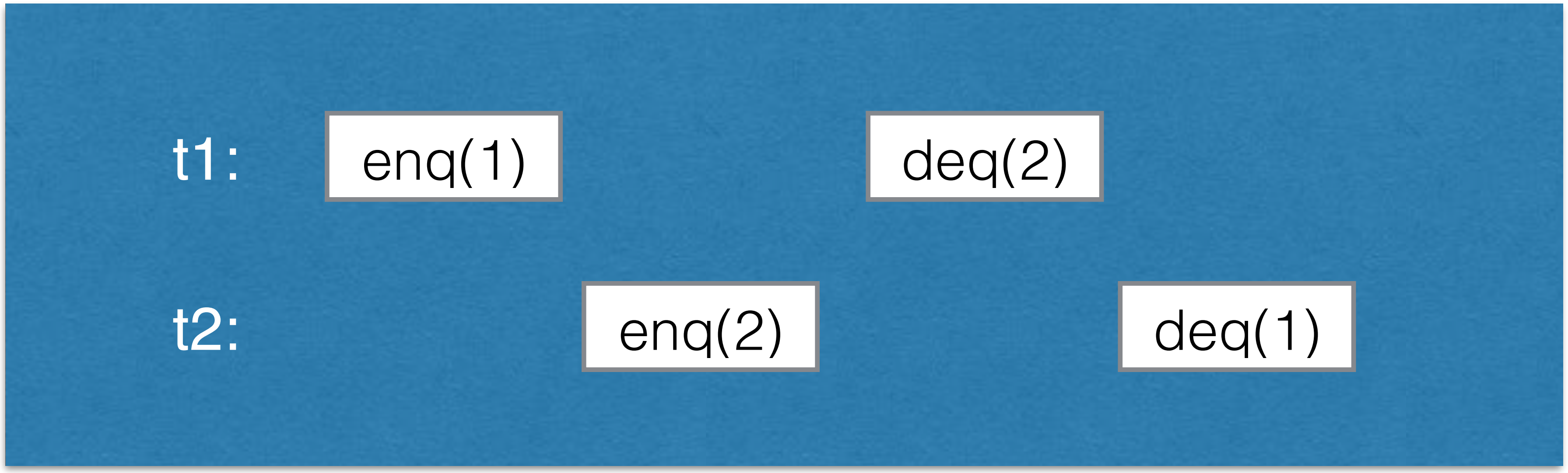
Already present in some shared-memory consistency conditions (not in our form of choice)

- Partition a history into a set of local histories
- Require linearizability per local history

no global witness

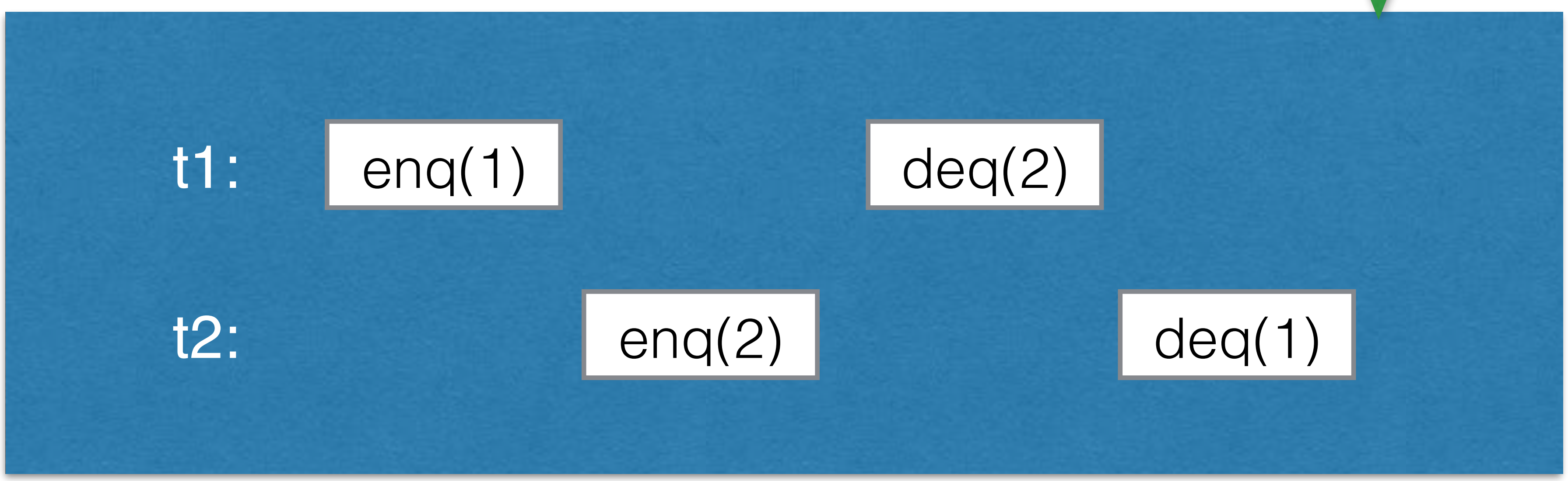
Local sequential consistency… is also possible

Local Linearizability (queue) example

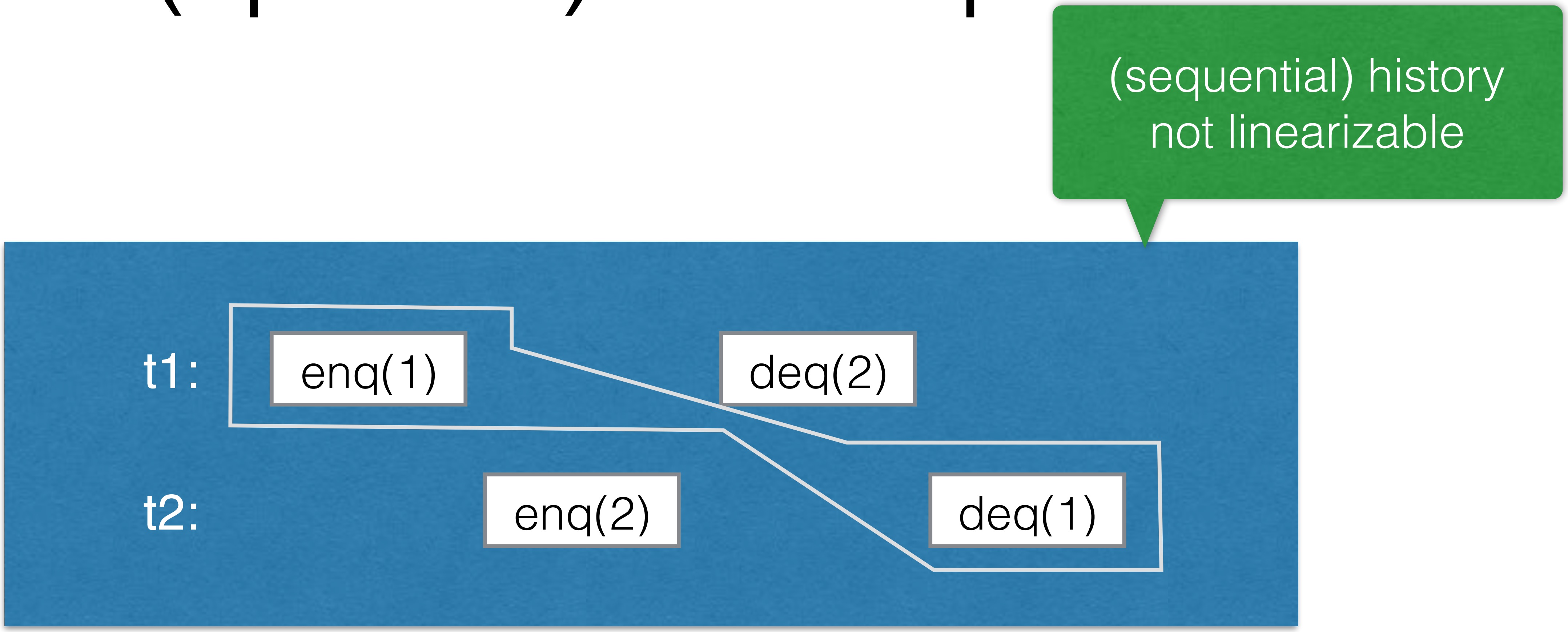


Local Linearizability (queue) example

(sequential) history not linearizable

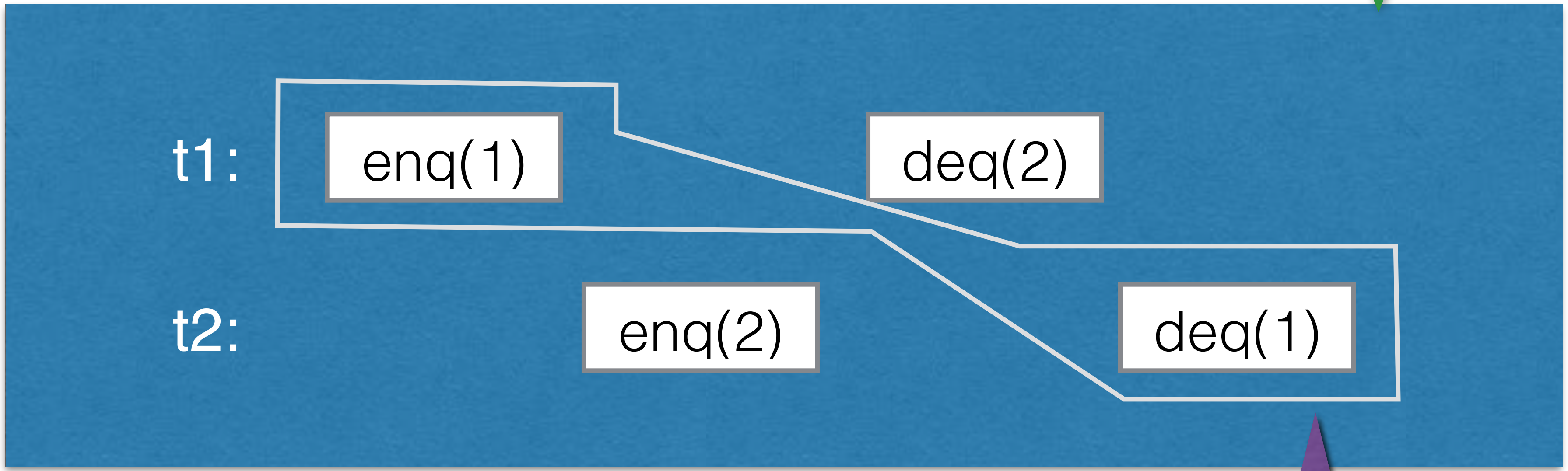


Local Linearizability (queue) example



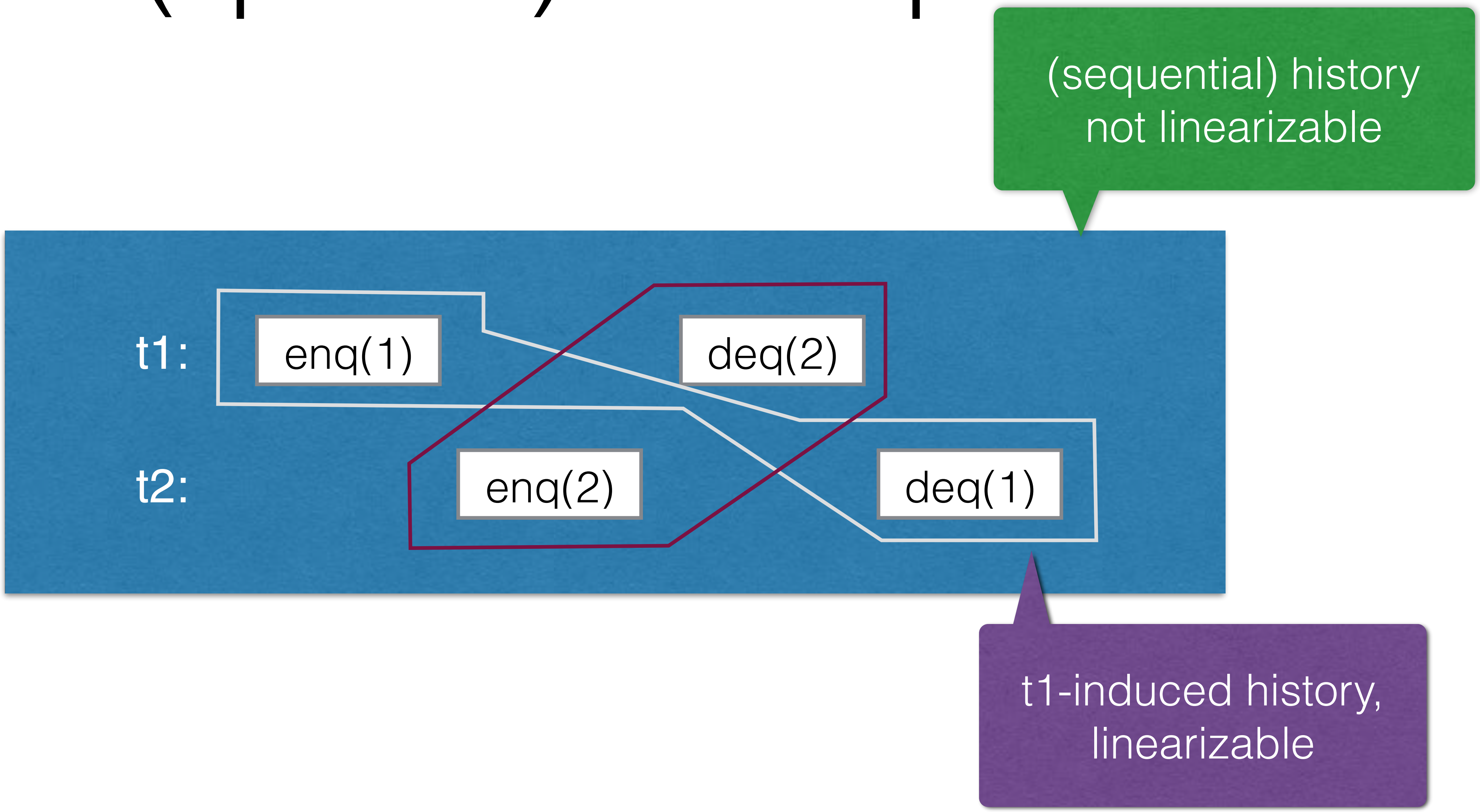
Local Linearizability (queue) example

(sequential) history not linearizable

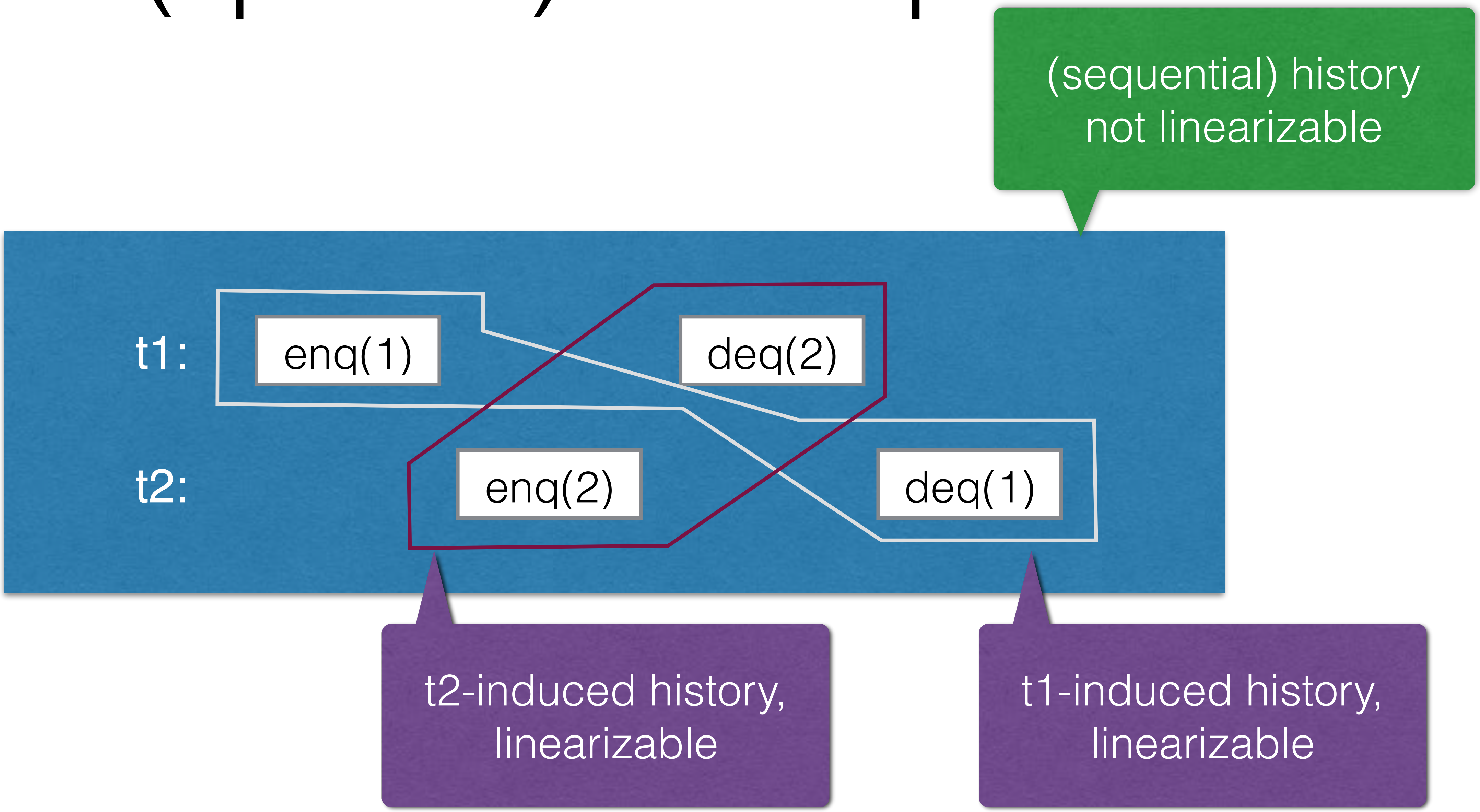


t1-induced history, linearizable

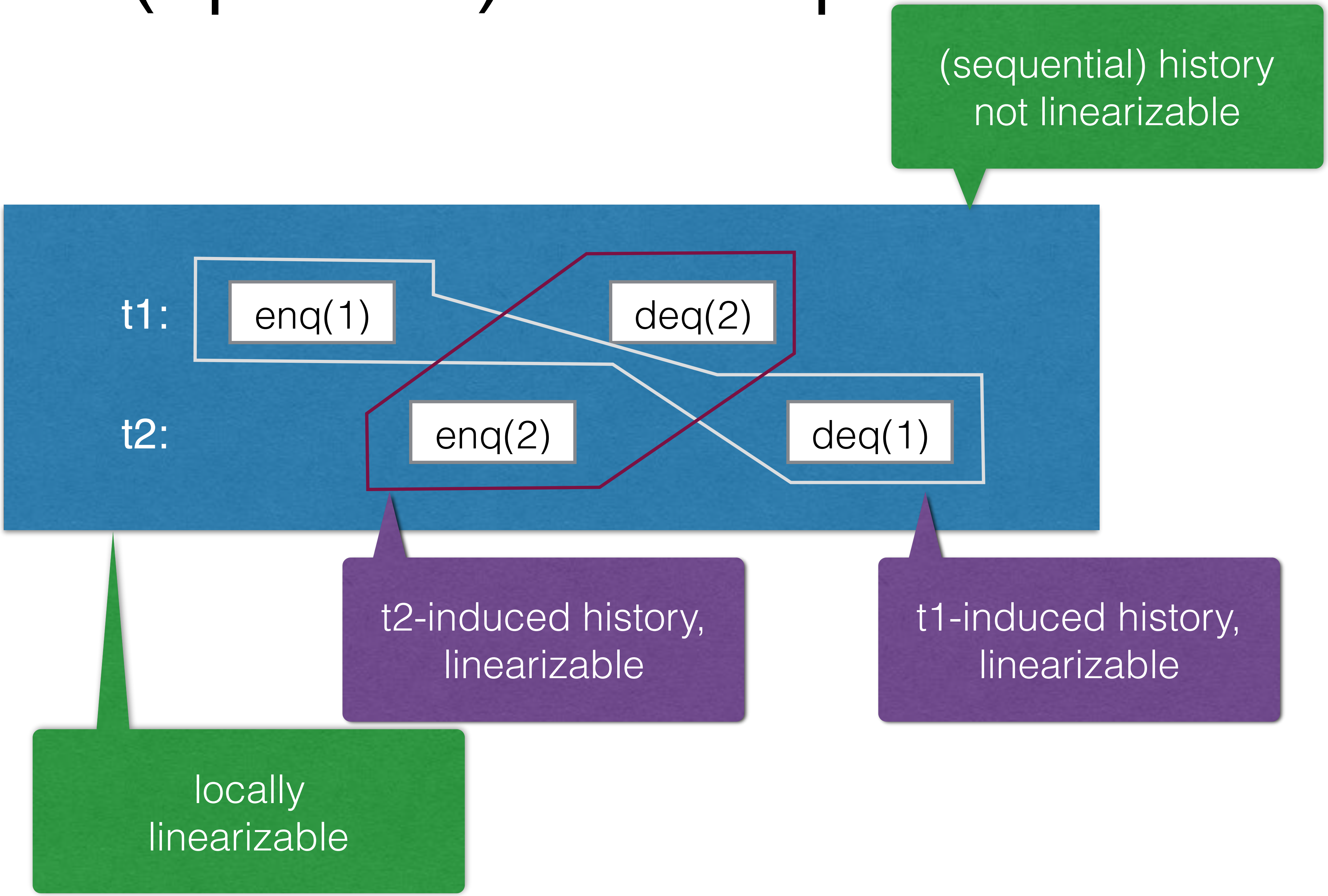
Local Linearizability (queue) example



Local Linearizability (queue) example



Local Linearizability (queue) example



Local Linearizability (queue) definition

Local Linearizability (queue) definition

Queue signature $\Sigma = \{enq(x) \mid x \in V\} \cup \{deq(x) \mid x \in V\} \cup \{deq(empty)\}$

Local Linearizability (queue) definition

Queue signature $\Sigma = \{enq(x) \mid x \in V\} \cup \{deq(x) \mid x \in V\} \cup \{deq(empty)\}$

For a history **h** with a thread T, we put

$$I_T = \{enq(x)^T \in \mathbf{h} \mid x \in V\}$$

$$O_T = \{deq(x)^{T'} \in \mathbf{h} \mid enq(x)^T \in I_T\} \cup \{deq(empty)\}$$

Local Linearizability (queue) definition

Queue signature $\Sigma = \{enq(x) \mid x \in V\} \cup \{deq(x) \mid x \in V\} \cup \{deq(empty)\}$

For a history **h** with a thread T, we put

$$I_T = \{enq(x)^T \in \mathbf{h} \mid x \in V\}$$

in-methods of thread T are enqueues performed by thread T

$$O_T = \{deq(x)^{T'} \in \mathbf{h} \mid enq(x)^T \in I_T\} \cup \{deq(empty)\}$$

Local Linearizability (queue) definition

Queue signature $\Sigma = \{enq(x) \mid x \in V\} \cup \{deq(x) \mid x \in V\} \cup \{deq(empty)\}$

For a history **h** with a thread T, we put

$$I_T = \{enq(x)^T \in \mathbf{h} \mid x \in V\}$$

$$O_T = \{deq(x)^{T'} \in \mathbf{h} \mid enq(x)^T \in I_T\} \cup \{deq(empty)\}$$

in-methods of thread T are enqueues performed by thread T

out-methods of thread T are dequeues (performed by any thread) corresponding to enqueues that are in-methods

Local Linearizability (queue) definition

Queue signature $\Sigma = \{enq(x) \mid x \in V\} \cup \{deq(x) \mid x \in V\} \cup \{deq(empty)\}$

For a history **h** with a thread T, we put

$$I_T = \{enq(x)^T \in \mathbf{h} \mid x \in V\}$$

in-methods of thread T are enqueues performed by thread T

$$O_T = \{deq(x)^{T'} \in \mathbf{h} \mid enq(x)^T \in I_T\} \cup \{deq(empty)\}$$

out-methods of thread T are dequeues (performed by any thread) corresponding to enqueues that are in-methods

h is locally linearizable iff every thread-induced history

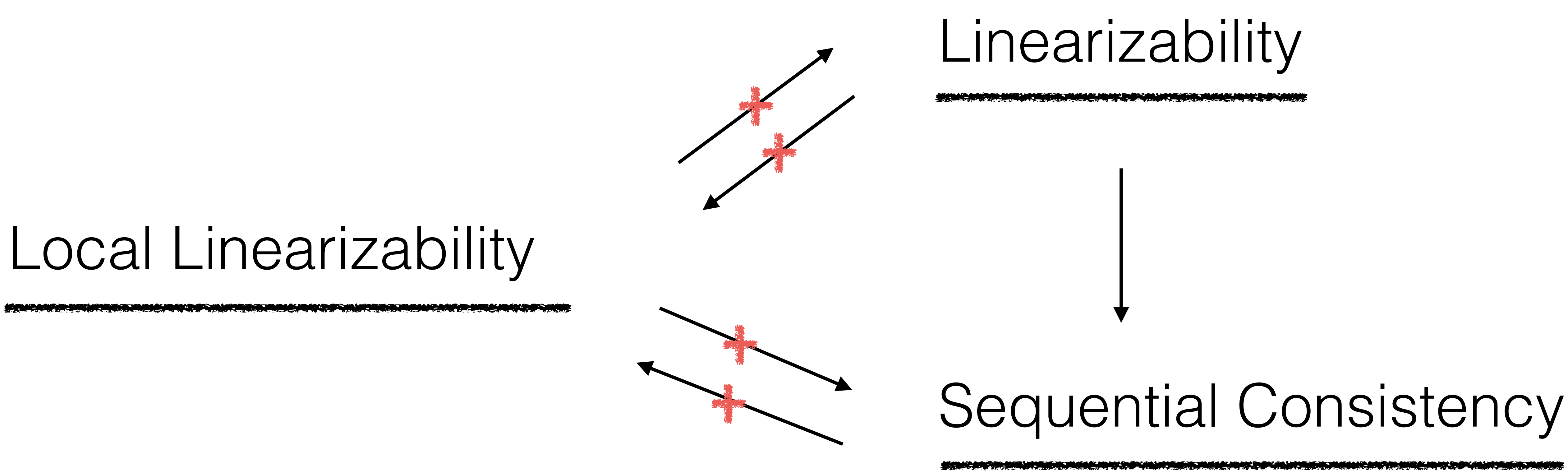
$$\mathbf{h}_T = \mathbf{h} \mid (I_T \cup O_T)$$

is linearizable.

Where do we stand?

Where do we stand?

In general



Where do we stand?

For queues (and most container-type data structures)

Linearizability

Local Linearizability

Sequential Consistency

Properties

Properties

Local linearizability is compositional

Properties

Local linearizability is compositional

like linearizability
unlike sequential consistency

Properties

Local linearizability is compositional

like linearizability
unlike sequential consistency

h (over multiple objects) is locally linearizable
iff
each per-object subhistory of **h** is locally linearizable

Properties

Local linearizability is compositional

like linearizability
unlike sequential consistency

h (over multiple objects) is locally linearizable
iff
each per-object subhistory of **h** is locally linearizable

Local linearizability is modular /
"decompositional"

Properties

Local linearizability is compositional

like linearizability
unlike sequential consistency

h (over multiple objects) is locally linearizable
iff
each per-object subhistory of **h** is locally linearizable

Local linearizability is modular /
"decompositional"

uses decomposition into smaller
histories, by definition

Properties

Local linearizability is compositional

like linearizability
unlike sequential consistency

h (over multiple objects) is locally linearizable
iff
each per-object subhistory of **h** is locally linearizable

Local linearizability is modular / "decompositional"

uses decomposition into smaller histories, by definition

↓

may allow for modular verification

Generic Implementations

Generic Implementations

Your favorite linearizable data structure implementation

Generic Implementations

Your favorite linearizable data structure implementation

Φ

Generic Implementations

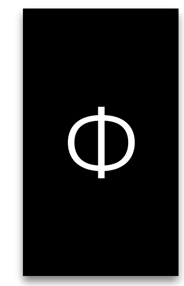
Your favorite linearizable data structure implementation

Φ

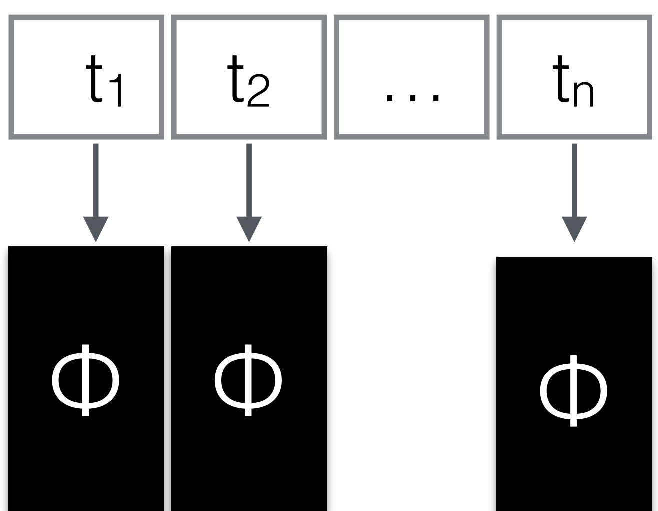
turns into a locally linearizable implementation by:

Generic Implementations

Your favorite linearizable data structure implementation

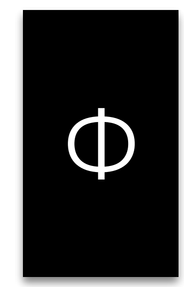


turns into a locally linearizable implementation by:

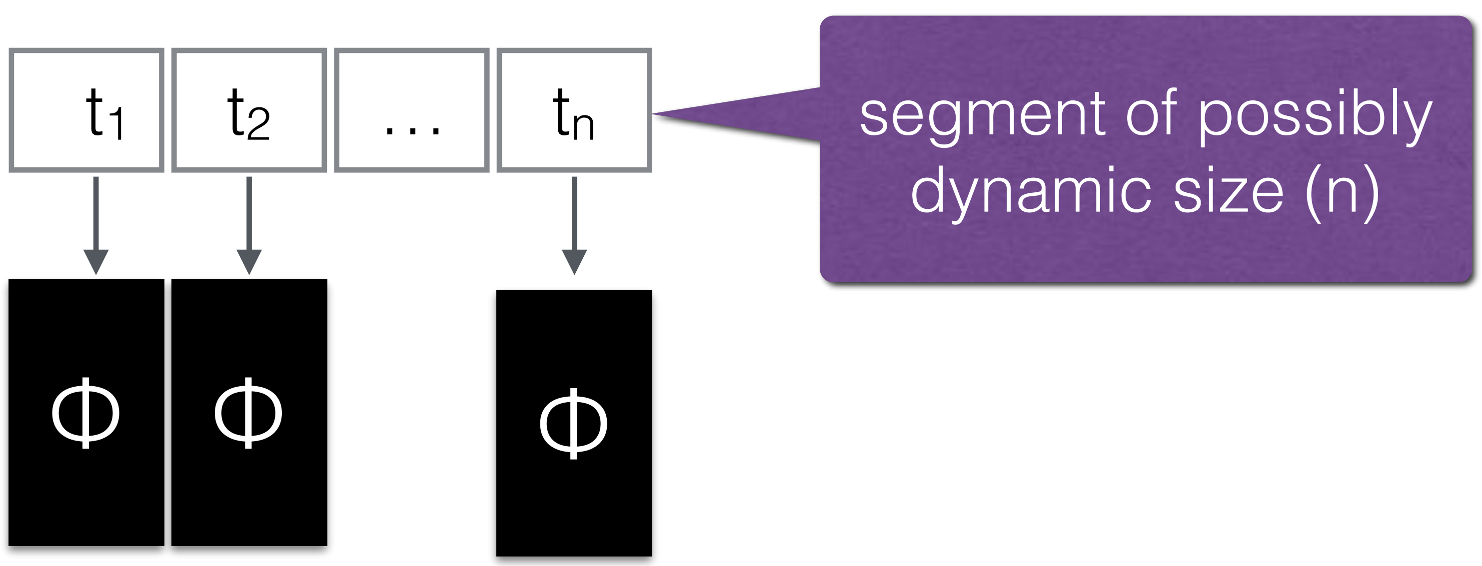


Generic Implementations

Your favorite linearizable data structure implementation

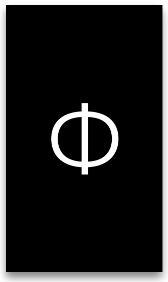


turns into a locally linearizable implementation by:

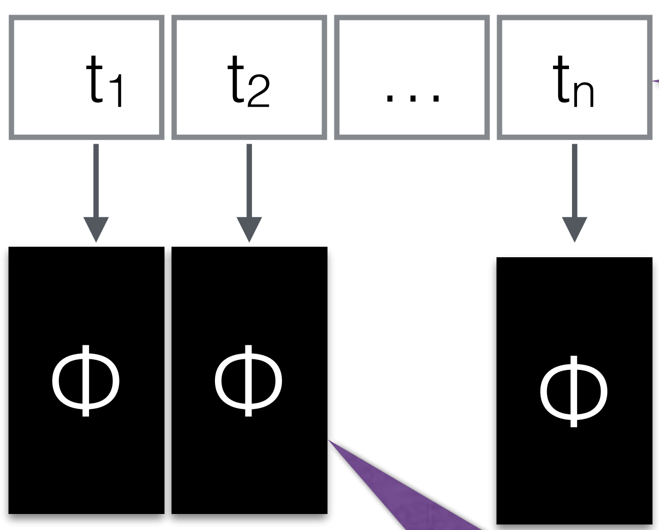


Generic Implementations

Your favorite linearizable data structure implementation



turns into a locally linearizable implementation by:



segment of possibly dynamic size (n)

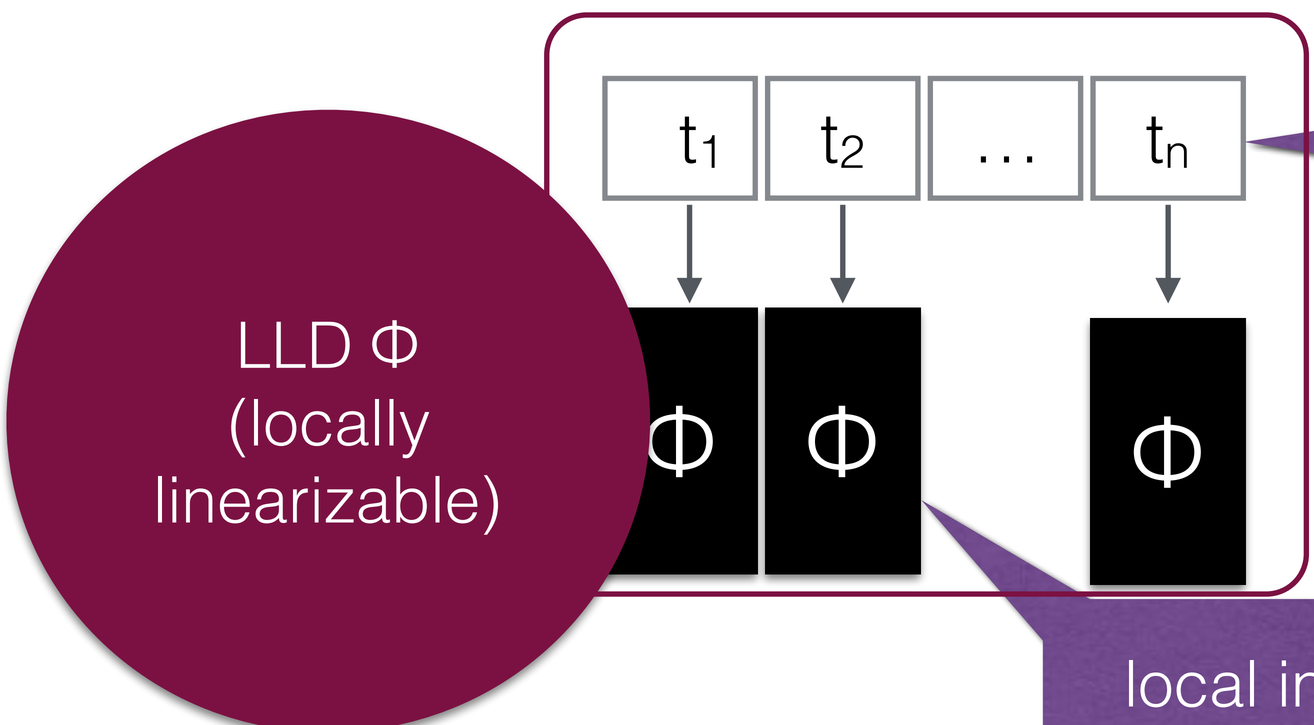
local inserts / global (randomly distributed) removes

Generic Implementations

Your favorite linearizable data structure implementation

Φ

turns into a locally linearizable implementation by:



segment of possibly dynamic size (n)

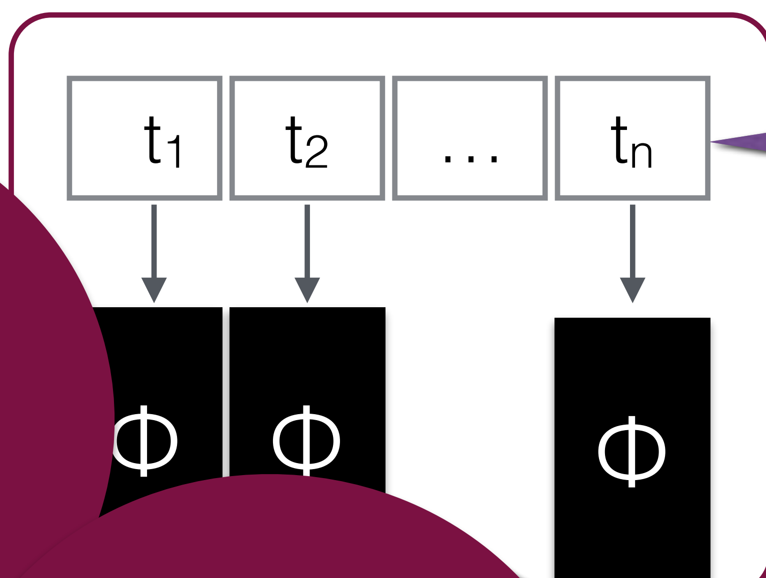
local inserts / global (randomly distributed) removes

Generic Implementations

Your favorite linearizable data structure implementation

Φ

turns into a locally linearizable implementation by:



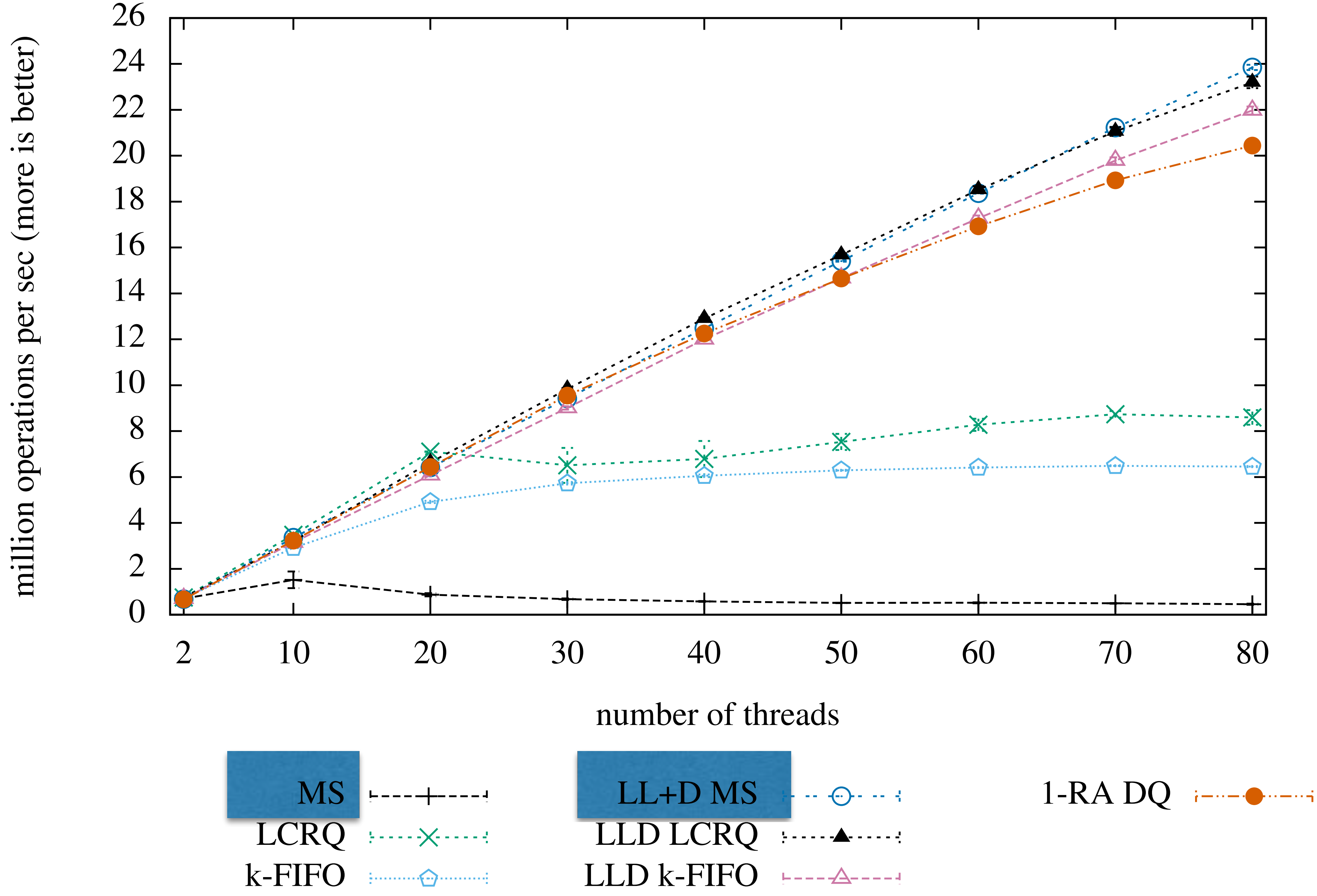
segment of possibly dynamic size (n)

local inserts / global (randomly distributed) removes

LLD Φ
(locally linearizable)

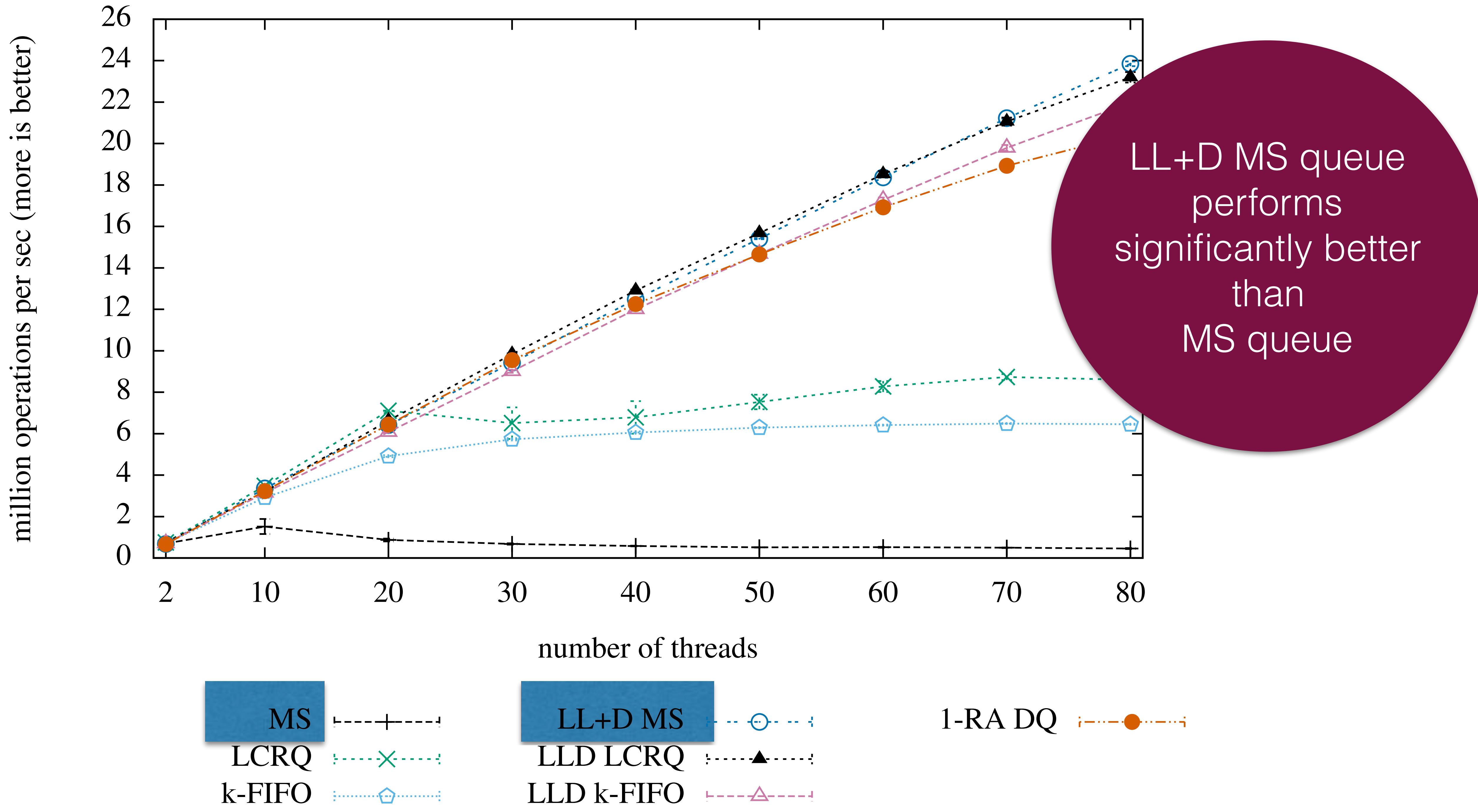
LL+D Φ
(also pool linearizable)

Performance



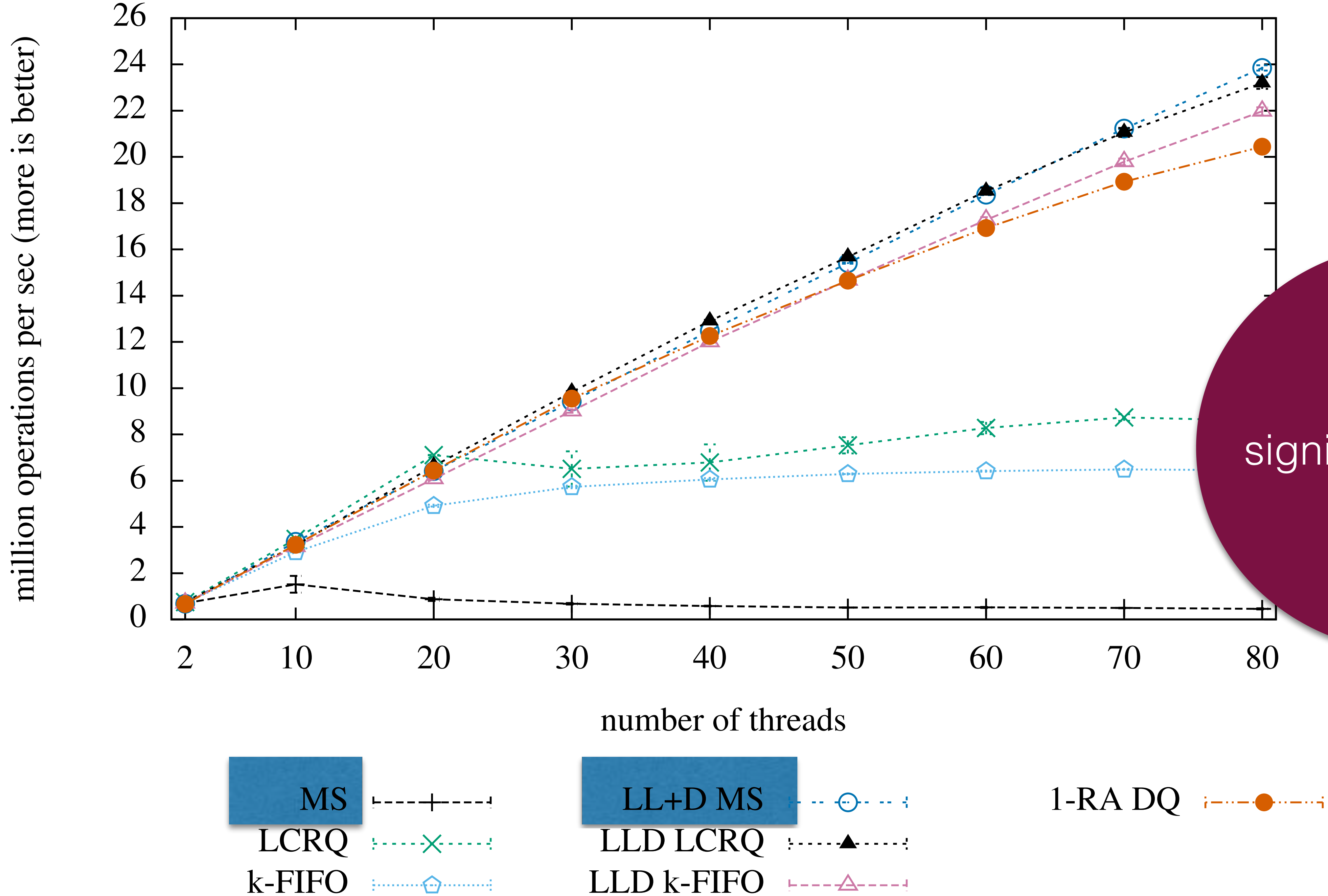
(a) Queues, LL queues, and "queue-like" pools

Performance



(a) Queues, LL queues, and “queue-like” pools

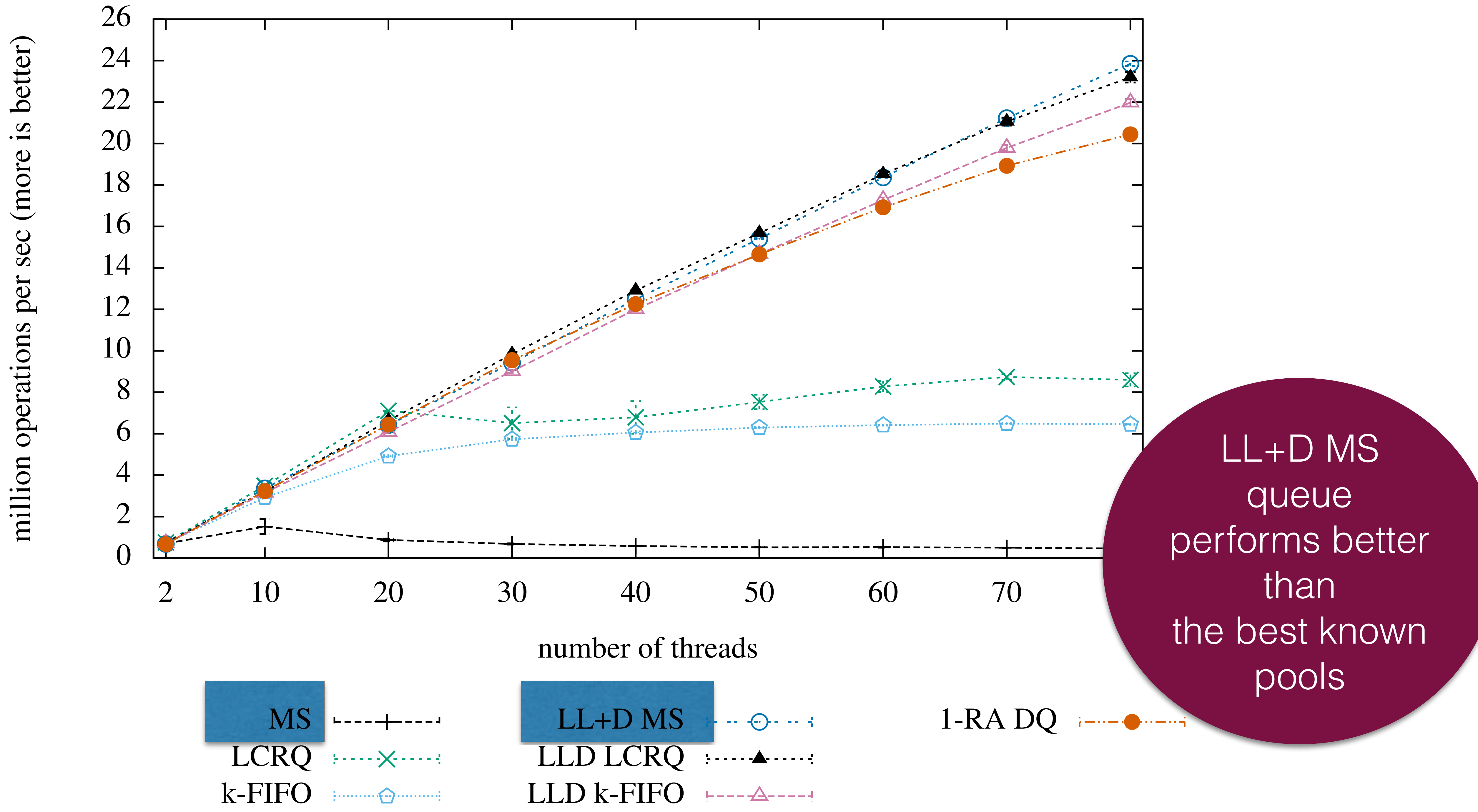
Performance



(a) Queues, LL queues, and "queue-like" pools

LLD Φ performs significantly better than Φ

Performance

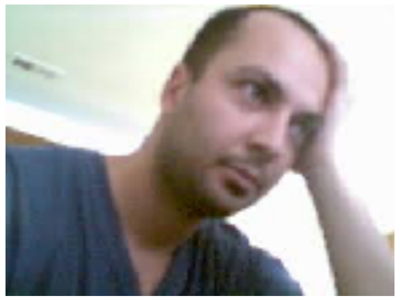


(a) Queues, LL queues, and "queue-like" pools

Thank You!

and many thanks to my dear coauthors

Andreas Holzer Google
UNIVERSITY OF TORONTO

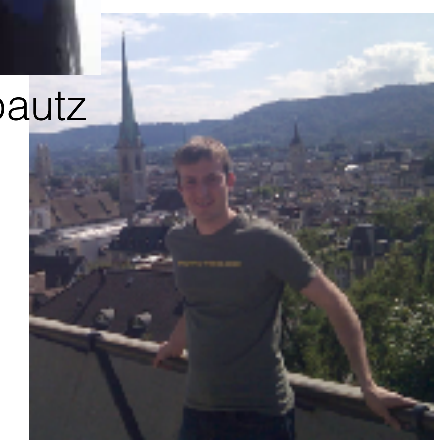


Ali Sezgin UNIVERSITY OF CAMBRIDGE

Hannes Payer
Google

Michael Lippautz

Tom Henzinger
IST AUSTRIA



Andreas Haas Google

Helmut Veith TU WIEN

Christoph Kirsch
UNIVERSITY of SALZBURG