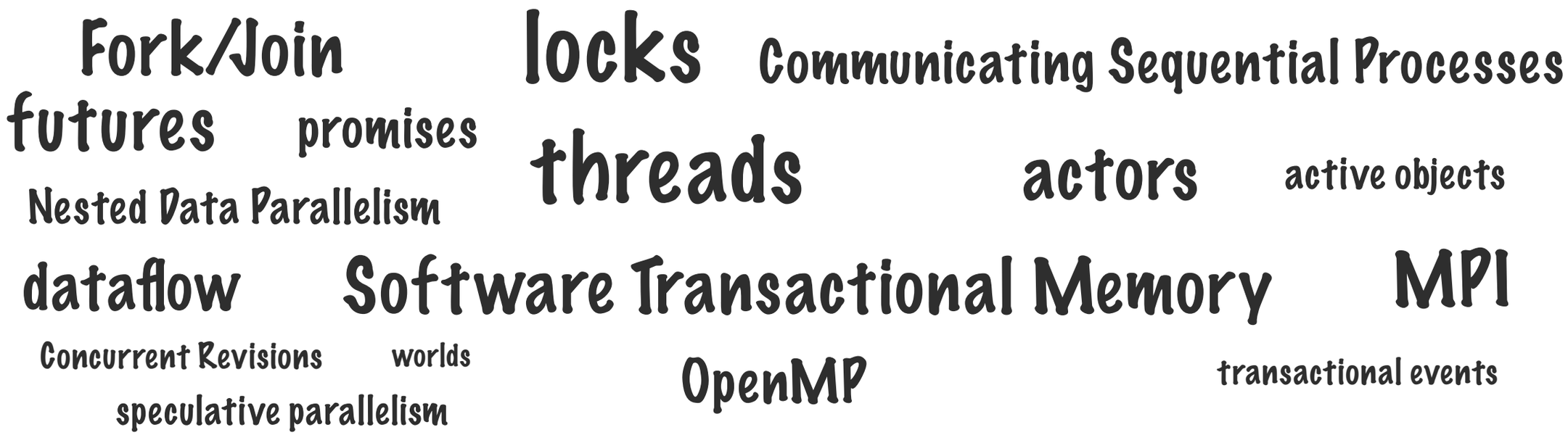
A Multi-Paradigm Concurrent Programming Model

Janwillem Swalens

Promotors:
Prof. Dr. Wolfgang De Meuter
Prof. Dr. Joeri De Koster

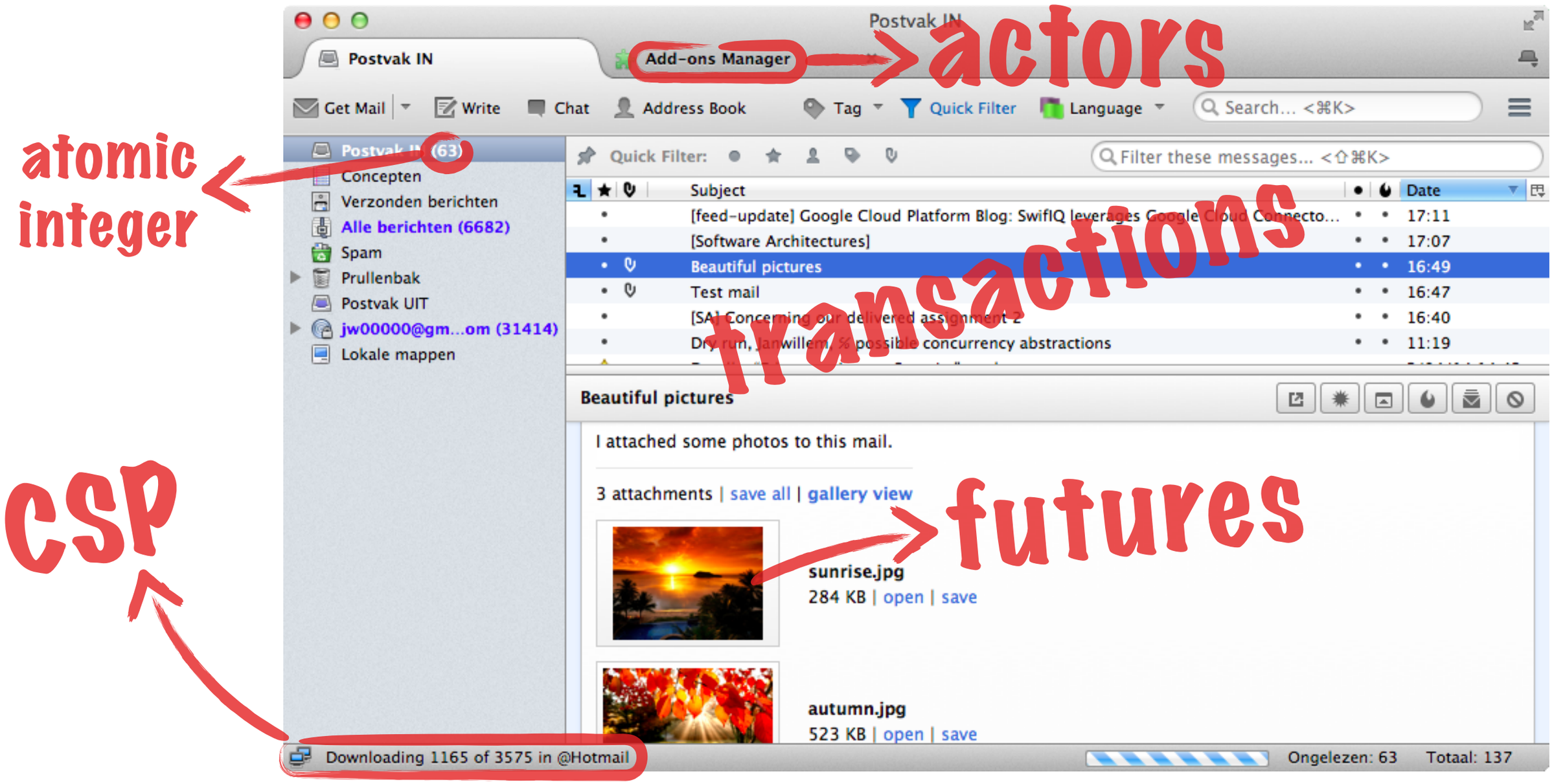
Concurrency is necessary but difficult

Concurrency bugs are frequent,
difficult to reproduce, and difficult to debug

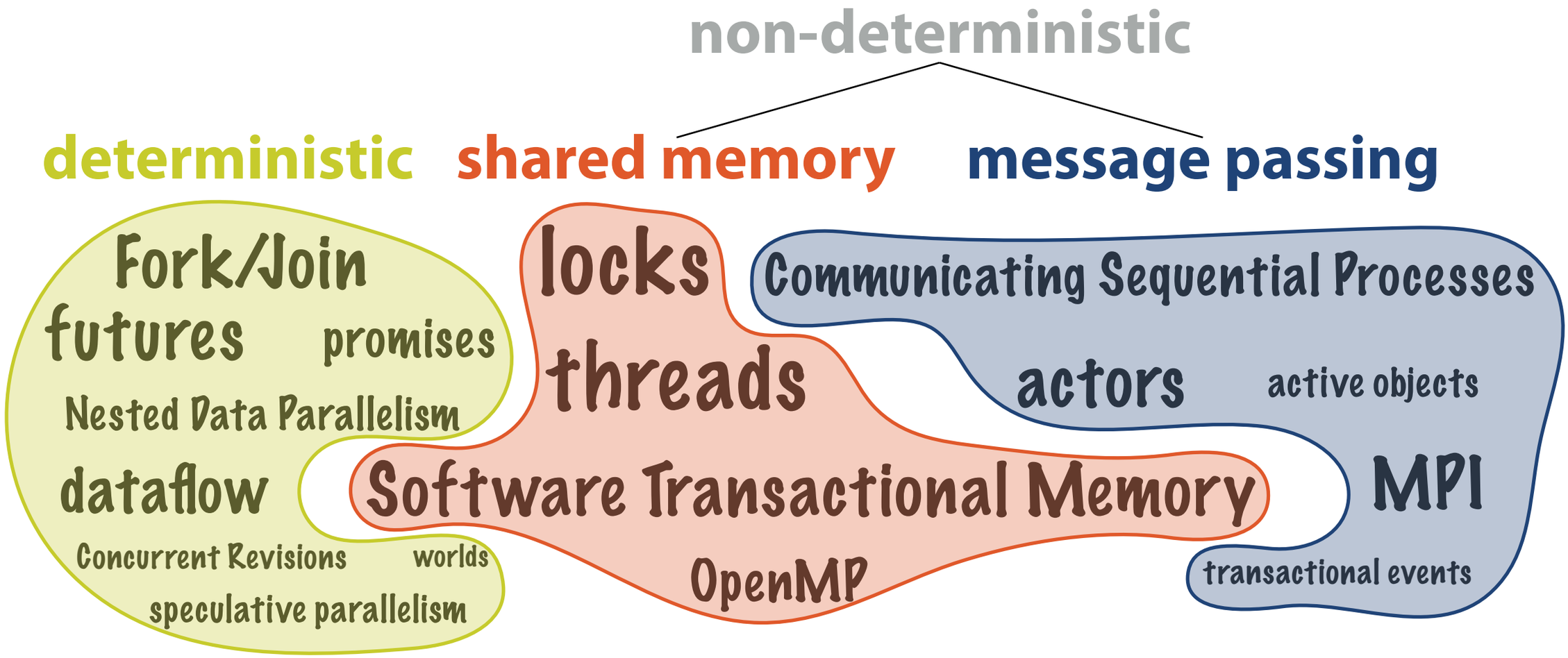


Zhou, Neamtiu, and Gupta (2015). *Predicting Concurrency Bugs: How Many, What Kind and Where Are They?* (EASE'15)
Godefroid and Nagappan (2008). *Concurrency at Microsoft – An Exploratory Survey*

Different concurrency models target different use cases

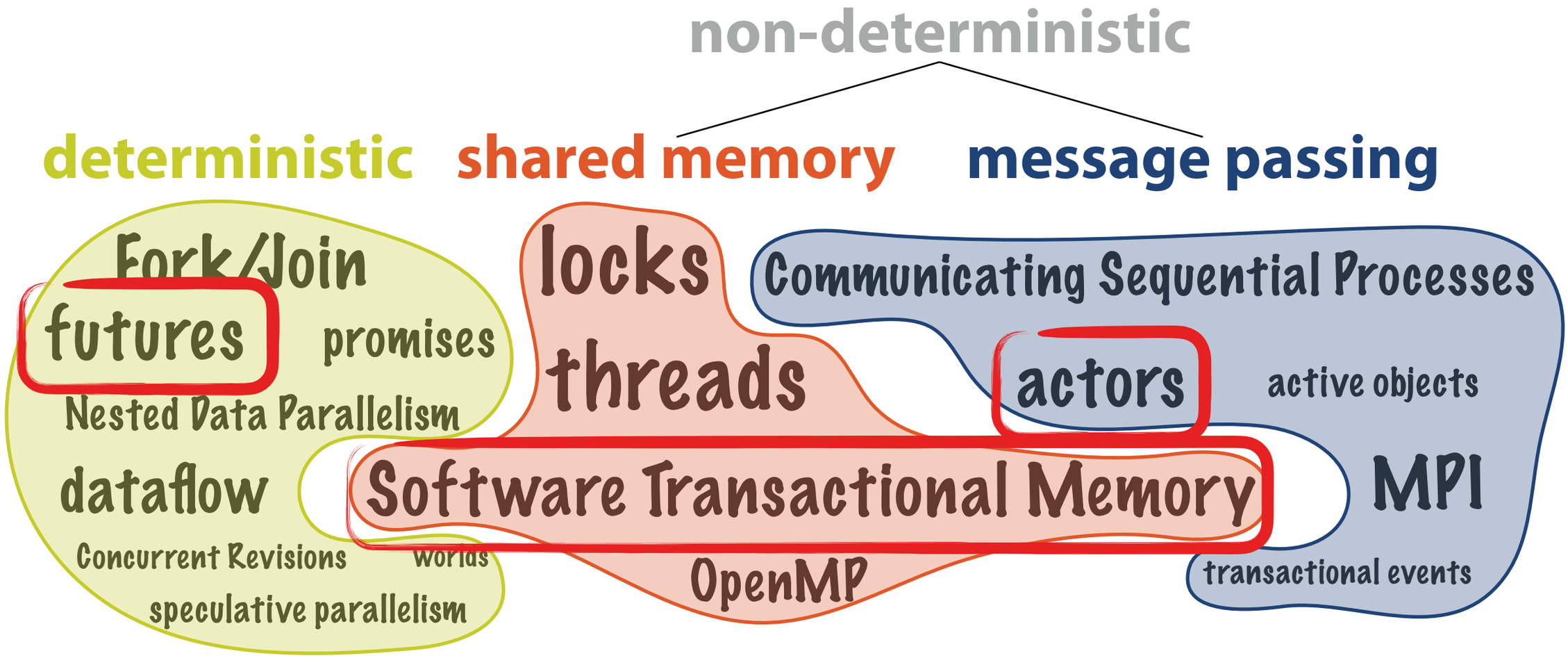


There are many different concurrency models



Van Roy, Haridi (2004). *Concepts, Techniques, and Models of Computer Programming* (The MIT Press)

There are many different concurrency models



Van Roy, Haridi (2004). *Concepts, Techniques, and Models of Computer Programming* (The MIT Press)

Futures

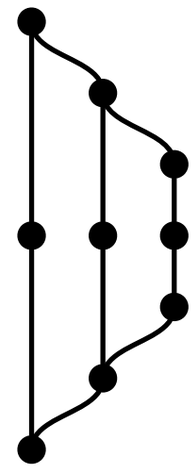
```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fib (- n 1))
          b (fib (- n 2))]
      (+ a b))))
```

Futures

```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fork (fib (- n 1)))
          b (fork (fib (- n 2)))]
      (+ (join a) (join b)))))
```

Guarantee:

Det **determinacy**



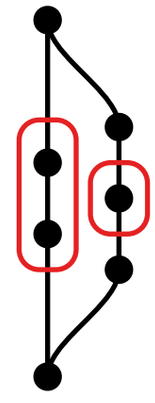
Transactions

```
(def checking (ref 100))
(def savings  (ref 500))
(fork
  (atomic
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(fork
  (atomic
    (println “You own €” (+ (deref checking)
                            (deref savings)))))
```

Guarantees:

Iso **isolation** (e.g. serializability, snapshot isolation)

Pro **progress** (e.g. deadlock freedom)



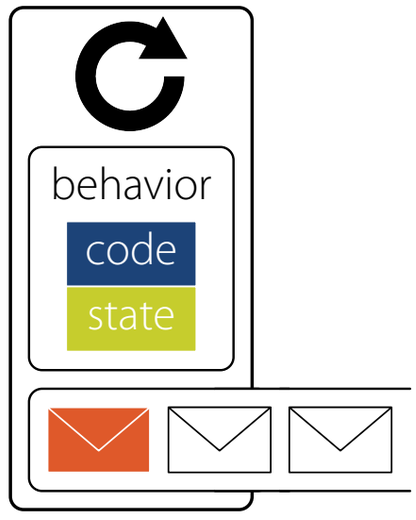
Actors

```
(def airline-behavior
  (behavior [flights]
    [orig dest n]
    (let [flight   (search-flight flights orig dest)
          flights' (reserve flights flight)]
      (become airline-behavior flights'))))

(def air-canada
  (spawn airline-behavior
    {"AC854" {:orig "YVR" :dest "LHR" :seats 211}
     "AC855" {:orig "LHR" :dest "YVR" :seats 211}}))

(send air-canada "LHR" "YVR" 2)
```

Guarantees:

ITP **isolated turn principle**

DLF **deadlock freedom**



Formalization

$\mathrm{L_f}$

Program state	p	::=	$\langle \mathrm{T} \rangle$
Tasks	$\mathrm{T} \subset \mathsf{Task}$		
Task	$\mathsf{task} \in \mathsf{Task}$	::=	$\langle f, e \rangle$

$\mathrm{L_t}$

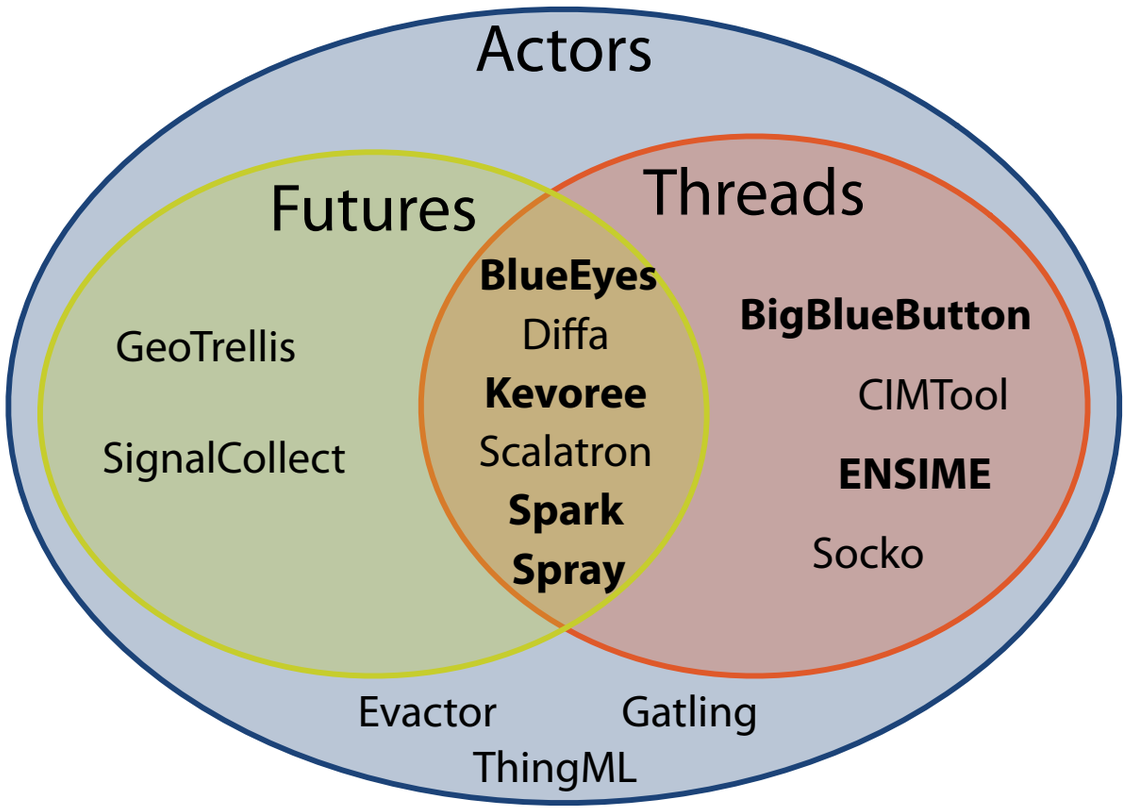
Program state	p	::=	$\langle \mathrm{T}, \tau, \sigma \rangle$
Task	$\mathsf{task} \in \mathsf{Task}$	::=	$\langle f, e, \mathsf{n}^? \rangle$
Transactions	$\tau : \mathsf{TransactionNumber} \rightharpoonup \mathsf{Transaction}$		
snapshot, local store	$\sigma, \overleftarrow{\sigma}, \delta : \mathrm{TVar} \rightharpoonup \mathrm{Value}$		
Transaction	$\mathsf{tx} \in \mathsf{Transaction}$	::=	$\langle \circ, \overleftarrow{\sigma}, \overleftarrow{e}, \delta \rangle$
Transaction id	$\mathsf{n} \in \mathsf{TransactionNumber}$	$=$	$\mathbb{N}^+$
Transaction state	$\circ$	::=	$\triangleright \mid \checkmark \mid \times$

$\mathrm{L_a}$

Program state	p	::=	$\langle \mathrm{A}, \mu \rangle$
Actors	$\mathrm{A} \subset \mathsf{Actor}$		
Inboxes	$\mu : \mathrm{Address} \rightharpoonup \overline{\mathsf{Message}}$		
Actor	$\mathsf{act} \in \mathsf{Actor}$	::=	$\langle a, e^?, \mathsf{beh} \rangle$
Behavior	$\mathsf{beh} \in \mathsf{Behavior}$	::=	$\langle b, \overline{v} \rangle$
Message	$\mathsf{msg} \in \mathsf{Message}$	::=	$\langle a_{\mathrm{from}}, a_{\mathrm{to}}, \overline{v} \rangle$

Uniform formalization of three separate models ☞ Chapter 2

Observation 1: programmers combine concurrency models



15 **Scala** programs with **actors**:

- 12/15 (80%) combine with **another model**
- 6/15 (40%) say they circumvent it where it is **"not a good fit"**

Tasharofi, Dinges, and Johnson (2013). *Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?* (ECOOP'13)

Observation 2: programming languages support many concurrency models

	Clojure	Scala	Java	Haskell	C++
Deterministic models					
Futures	✓	✓	✓	•	✓
Promises	✓	✓	✓	•	✓
Fork/Join	✓*	✓*	✓		•
Parallel collections	✓*	✓	✓	•	•
Dataflow	•	•	•	•	
Shared-memory models					
Threads	✓*	✓*	✓	✓	✓
Locks	✓*	✓*	✓	✓	✓
Atomic variables	✓	✓*	✓	✓	✓
Transactional memory	✓	•	•	✓	•
Message-passing models					
Actors	•	•	•	•	•
Channels	✓	✓	•	✓	•
Agents	✓				
# supported models	10	8	7	5	5

✓ built in
• library

Clojure has 6 concurrency models built in
(+ 4 through JVM)

Naive combinations lead to problems

Case study of Clojure

```
(swap! (fn [v] (send …)))
(send  (fn [v] (send …)))
(dosync        (send …))
(future        (send …))
(go            (send …))
```

```
(send (fn [v] (swap! …))
(send (fn [v] (send  …))
(send (fn [v] (dosync …))
(send (fn [v] (future …))
(send (fn [v] (promise …))
(send (fn [v] (go …))
```

Races

outer \ inner	Atom	Agent	STM	Future	Promise	Channel
Atom's swap!	✗	✗	✗	✗	✗	✗
Agent's action	✓	✓	✓	✓	✓	✓
STM's dosync	✗	✓	✓	✗	✗	✗
Future	✓	✓	✓	✓	✓	✓
CSP's go	✓	✓	✓	✓	✓	✓

Deadlocks

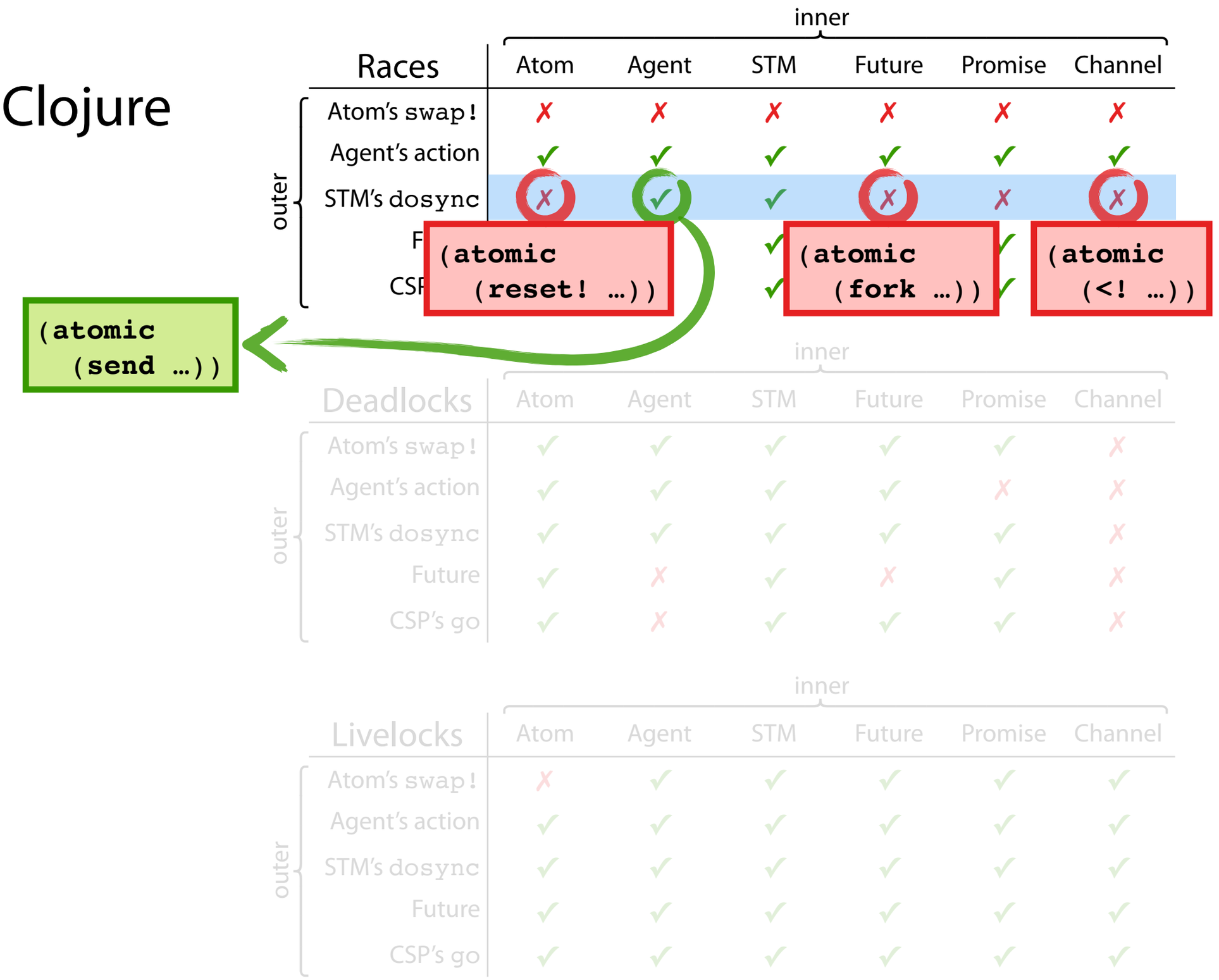
outer \ inner	Atom	Agent	STM	Future	Promise	Channel
Atom's swap!	✓	✓	✓	✓	✓	✗
Agent's action	✓	✓	✓	✓	✗	✗
STM's dosync	✓	✓	✓	✓	✓	✗
Future	✓	✗	✓	✗	✓	✗
CSP's go	✓	✗	✓	✓	✓	✗

Livelocks

outer \ inner	Atom	Agent	STM	Future	Promise	Channel
Atom's swap!	✗	✓	✓	✓	✓	✓
Agent's action	✓	✓	✓	✓	✓	✓
STM's dosync	✓	✓	✓	✓	✓	✓
Future	✓	✓	✓	✓	✓	✓
CSP's go	✓	✓	✓	✓	✓	✓

Naive combinations lead to problems

Case study of Clojure



Naive combinations lead to problems

3 common problems:

- spurious retries
  ⇒ races
- unexpected blocking
  ⇒ deadlocks
- unexpected retries
  ⇒ livelocks

Caused by operations that:

- retry
- block

Races	inner: Atom	Agent	STM	Future	Promise	Channel
outer: Atom's `swap!`	✗	✗	✗	✗	✗	✗
outer: Agent's action	✓	✓	✓	✓	✓	✓
outer: STM's `dosync`	✗	✓	✓	✗	✗	✗
outer: Future	✓	✓	✓	✓	✓	✓
outer: CSP's `go`	✓	✓	✓	✓	✓	✓

Deadlocks	inner: Atom	Agent	STM	Future	Promise	Channel
outer: Atom's `swap!`	✓	✓	✓	✓	✓	✗
outer: Agent's action	✓	✓	✓	✓	✗	✗
outer: STM's `dosync`	✓	✓	✓	✓	✓	✗
outer: Future	✓	✗	✓	✗	✓	✗
outer: CSP's `go`	✓	✗	✓	✓	✓	✗

Livelocks	inner: Atom	Agent	STM	Future	Promise	Channel
outer: Atom's `swap!`	✗	✓	✓	✓	✓	✓
outer: Agent's action	✓	✓	✓	✓	✓	✓
outer: STM's `dosync`	✓	✓	✓	✓	✓	✓
outer: Future	✓	✓	✓	✓	✓	✓
outer: CSP's `go`	✓	✓	✓	✓	✓	✓

Naive combinations lead to problems

We need to study each combination and how it affects the guarantees

Races	Atom	Agent	STM	Future	Promise	Channel
Atom's `swap!`	✗	✗	✗	✗	✗	✗
Agent's action	✓	✓	✓	✓	✓	✓
STM's `dosync`	✗	✓	✓	✗	✗	✗
Future	✓	✓	✓	✓	✓	✓
CSP's `go`	✓	✓	✓	✓	✓	✓

Deadlocks	Atom	Agent	STM	Future	Promise	Channel
Atom's `swap!`	✓	✓	✓	✓	✓	✗
Agent's action	✓	✓	✓	✓	✗	✗
STM's `dosync`	✓	✓	✓	✓	✓	✗
Future	✓	✗	✓	✗	✓	✗
CSP's `go`	✓	✗	✓	✓	✓	✗

Livelocks	Atom	Agent	STM	Future	Promise	Channel
Atom's `swap!`	✗	✓	✓	✓	✓	✓
Agent's action	✓	✓	✓	✓	✓	✓
STM's `dosync`	✓	✓	✓	✓	✓	✓
Future	✓	✓	✓	✓	✓	✓
CSP's `go`	✓	✓	✓	✓	✓	✓

We studied the combinations of futures, transactions, and actors

	Future	Transaction	Actor

Future	`(fork`
	`  (fork …)`
	`  (join …))`
	Nested futures
		`(fork`
		`  (atomic …))`
		Parallel transactions
			`(fork`
			`  (spawn …)`
			`  (send …)`
			`  (become …))`
			Communication in future

Transaction	`(atomic`
	`  (fork …)`
	`  (join …))`
	Parallelism in transaction
		`(atomic`
		`  (atomic …)`
		`  (ref …)`
		`  (deref …)`
		`  (ref-set …))`
		Nested transactions
			`(atomic`
			`  (spawn …)`
			`  (send …)`
			`  (become …))`
			Communication in transaction

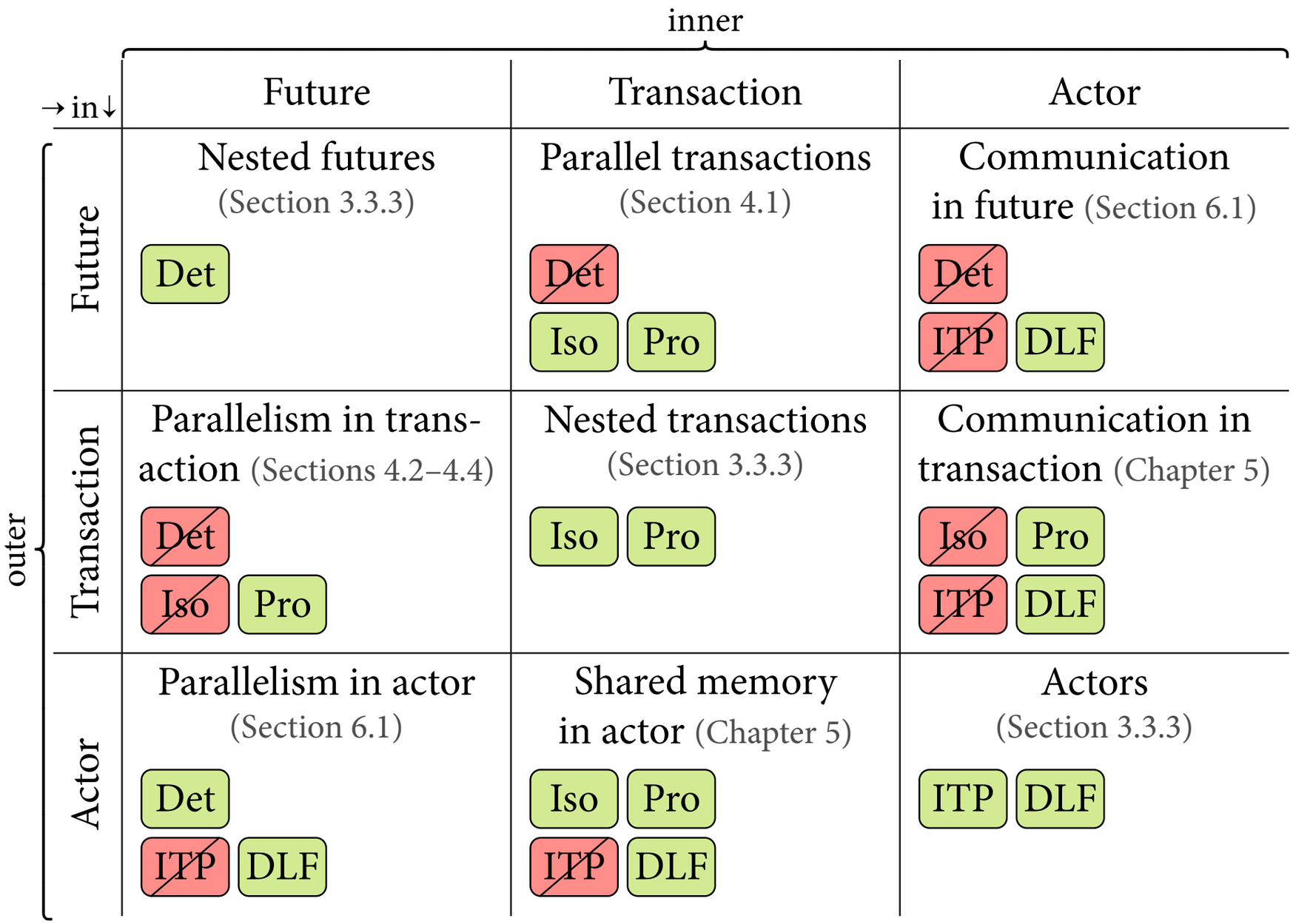
Actor	`(behavior [] []`
	`  (fork …)`
	`  (join …))`
	Parallelism in actor
		`(behavior [] []`
		`  (atomic …))`
		Shared memory in actor
			`(behavior [] []`
			`  (spawn …)`
			`  (send …)`
			`  (become …))`
			Actors

Goals

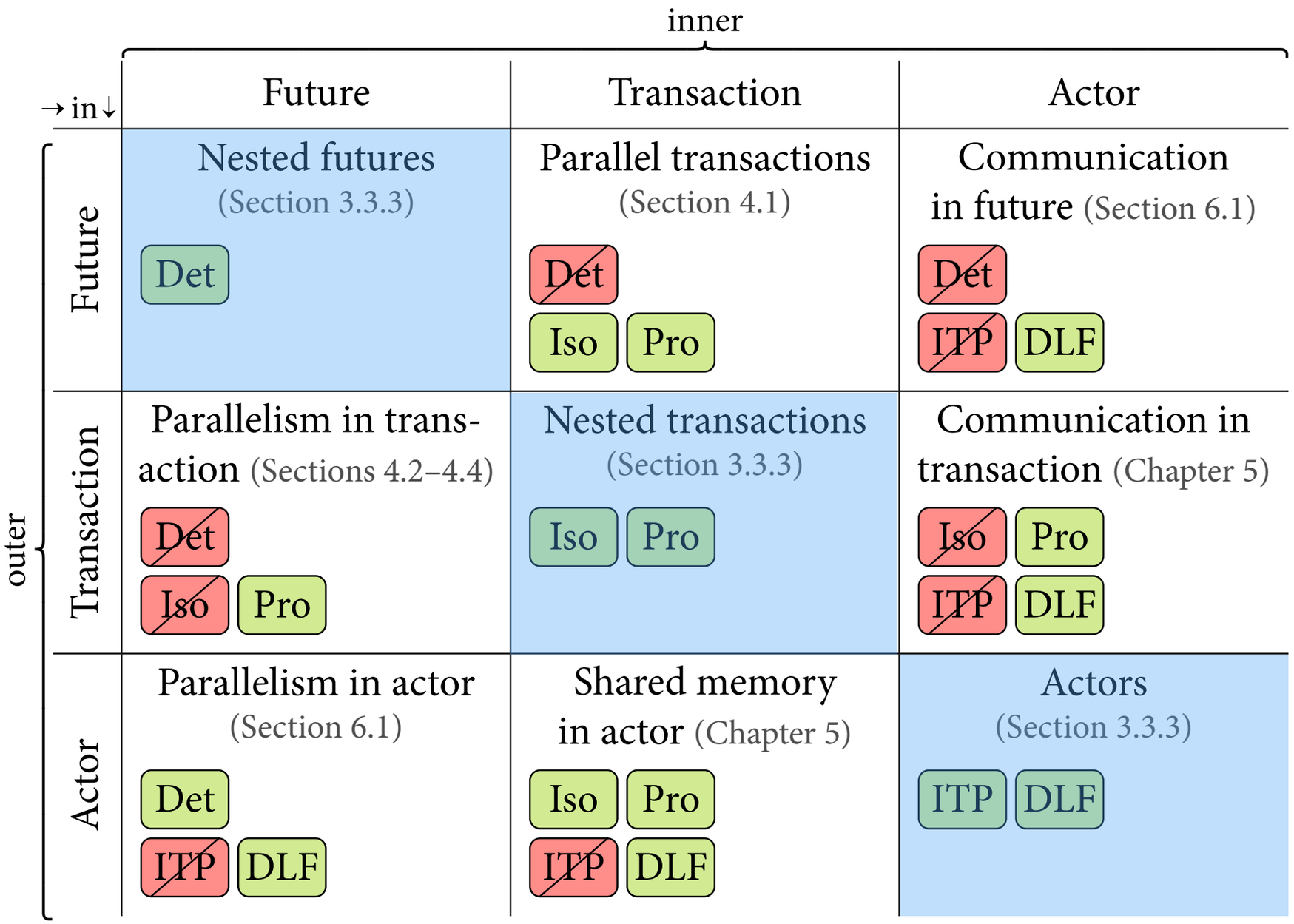
1. Separate models: backward compatibility

2. **Combinations: maintain guarantees of all models**
   If impossible: define a less restrictive guarantee

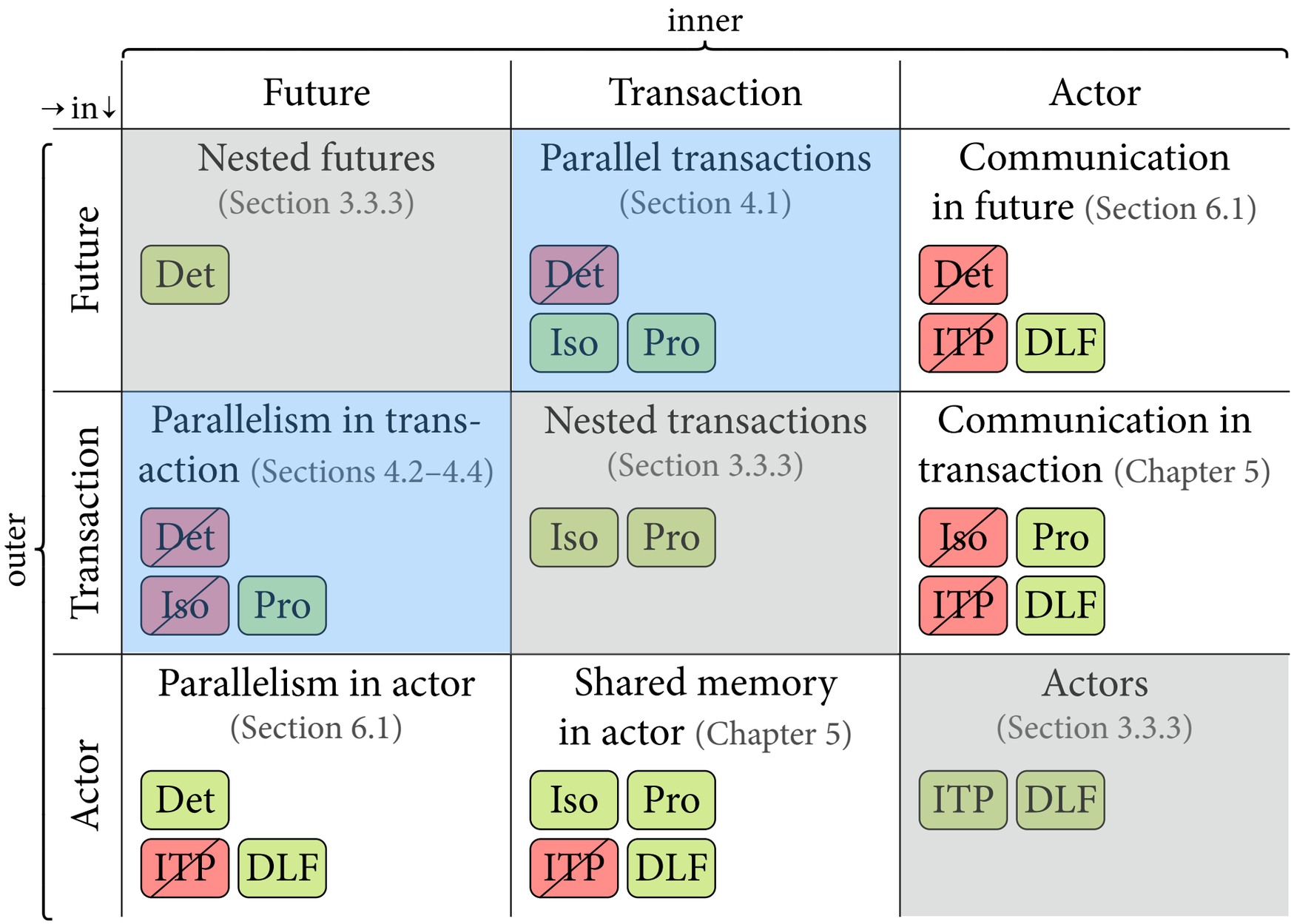
“Naive” combinations



Trivial combinations



Transactions + Futures



Motivation: Parallelism in Transaction

Application	Transaction length (mean # of instructions per tx)	Average time in transaction
Labyrinth	219,571	100%
Bayes	60,584	83%
Yada	9,795	100%
Vacation-high	3,223	86%
Genome	1,717	97%
Intruder	330	33%
Kmeans-high	117	7%
SSCA2	50	17%

parallelism *within* transaction

Labyrinth original:

```
(atomic
  (breadth-first-search …))
```

Labyrinth optimized:

```
(atomic
  (parallel-bfs …))

      (defn parallel-bfs […]
        (for […]
          (fork …)))
```

Minh, Chung, Kozyrakis, Olukotun (2008). *STAMP: Stanford Transactional Applications for Multi-Processing* (IISWC'08)

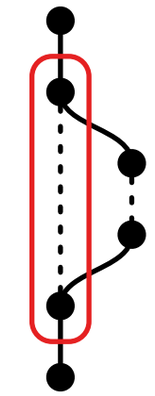
Problems when creating future in transaction

Impure languages (e.g. Clojure, ScalaSTM)

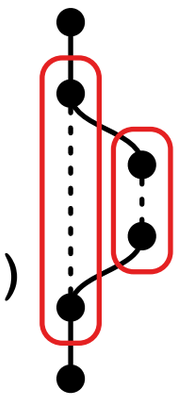
Tasks in transaction do not share context

⇒ **no access to transactional state**
or
⇒ **isolation broken** Iso

```
(atomic
  (fork
    (ref-set …)))
```



```
(atomic
  (fork
    (atomic
      (ref-set …))))
```

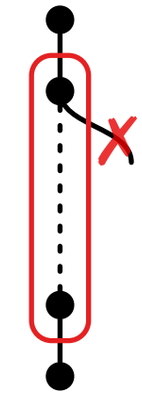


Pure languages (Haskell)

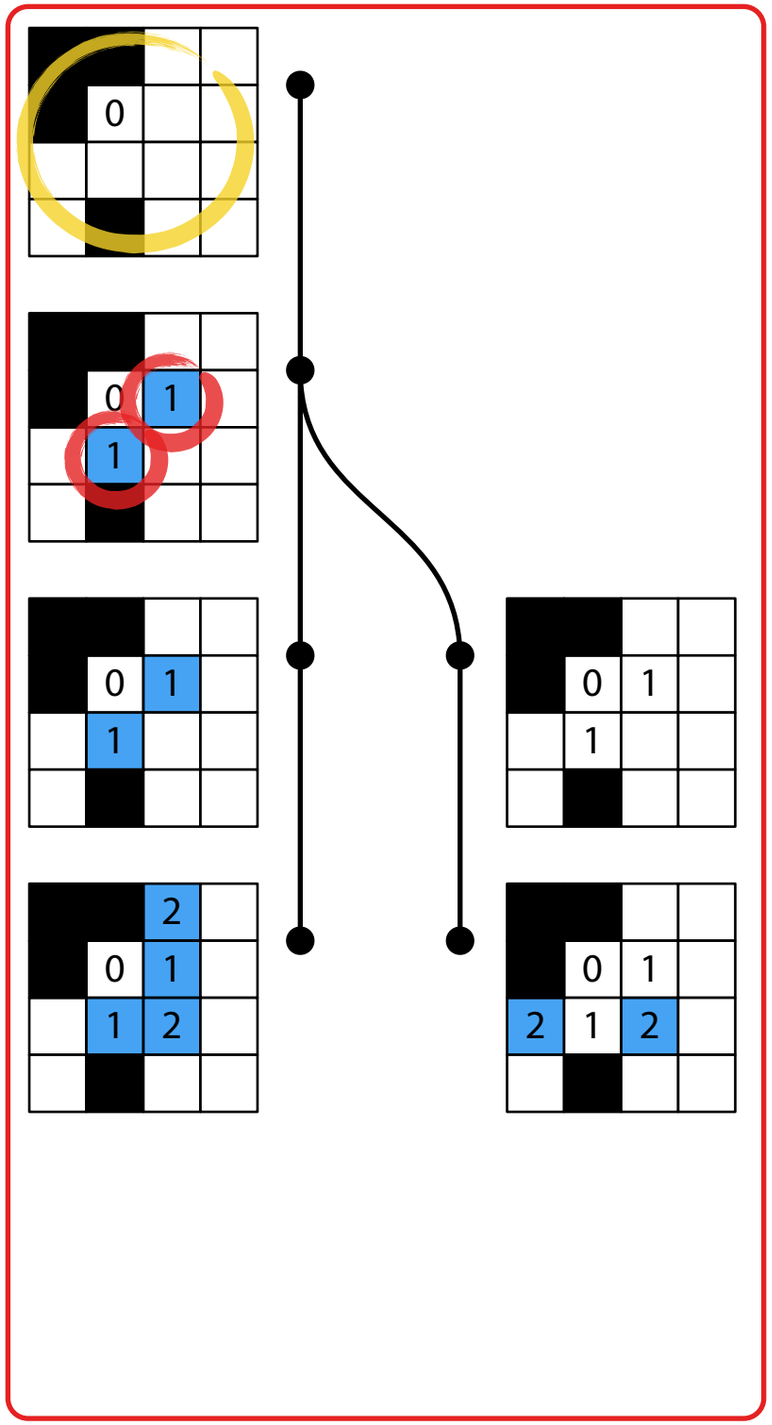
Tasks in transaction prohibited

⇒ isolation guaranteed but **parallelism limited**

```
atomically $
  do { forkIO … }
```

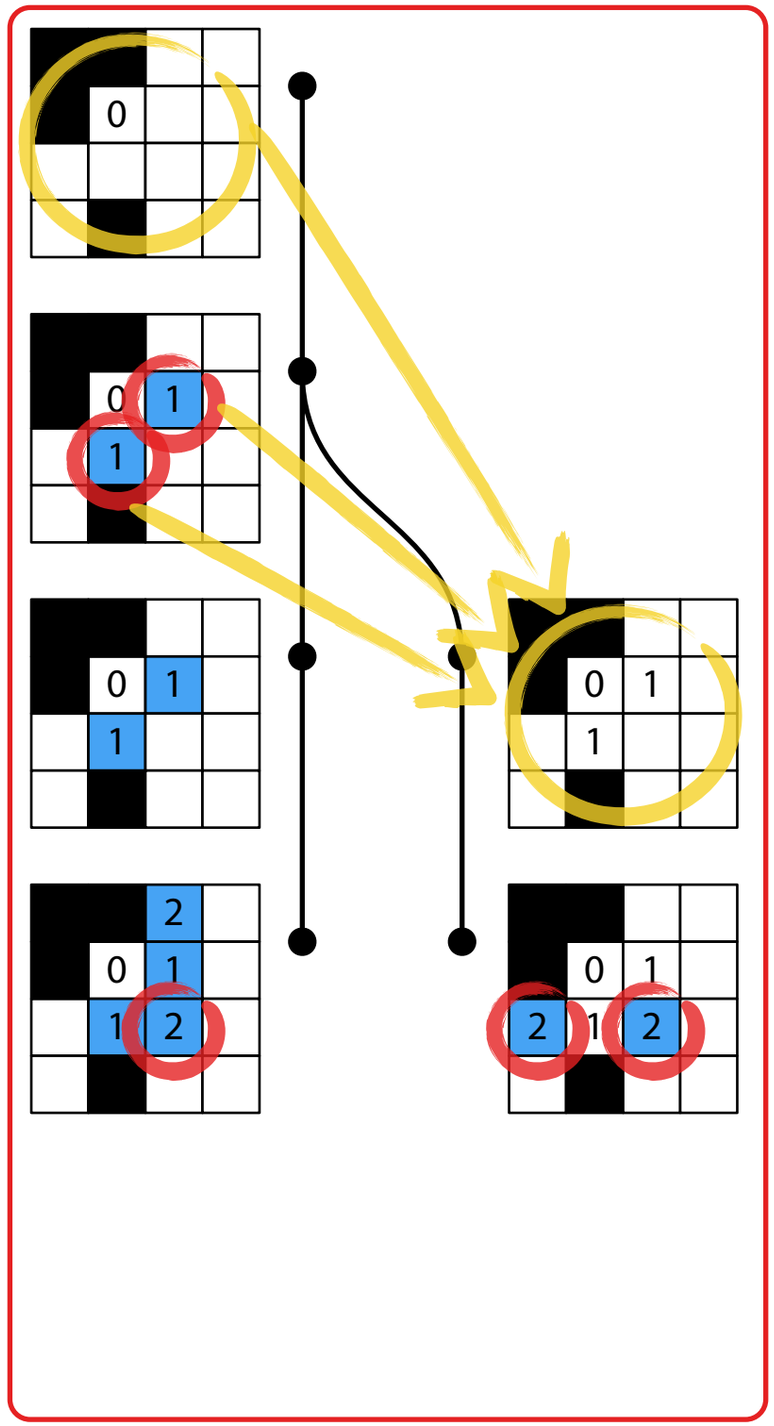


Transactional Futures




```
(atomic
  (ref-set … 1)
```

`fork` creates isolated task



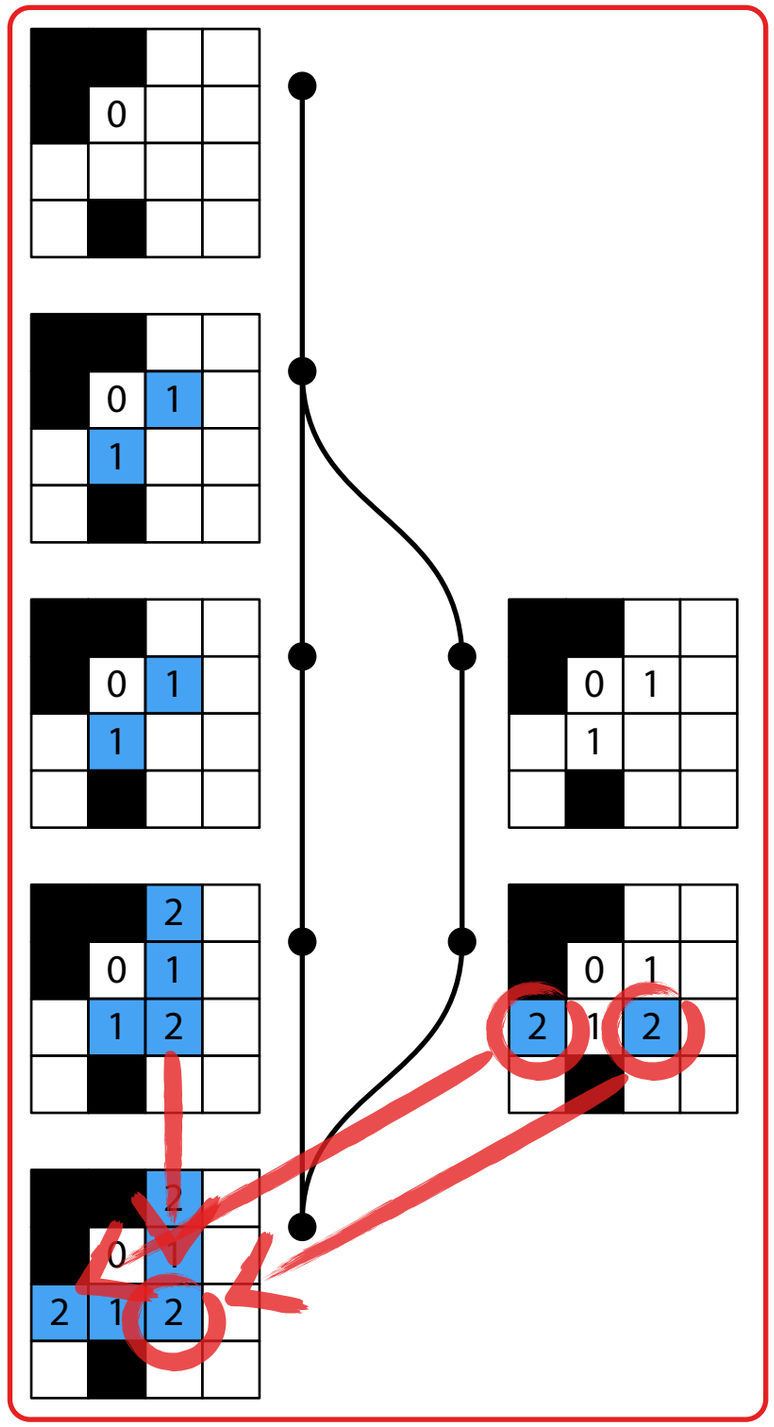

```
(atomic
  (ref-set … 1)
  (fork
    (ref-set … 2))
  (ref-set … 2)
```

Each transactional task contains:

snapshot: transactional state on creation

local store: local modifications

`join` merges changes



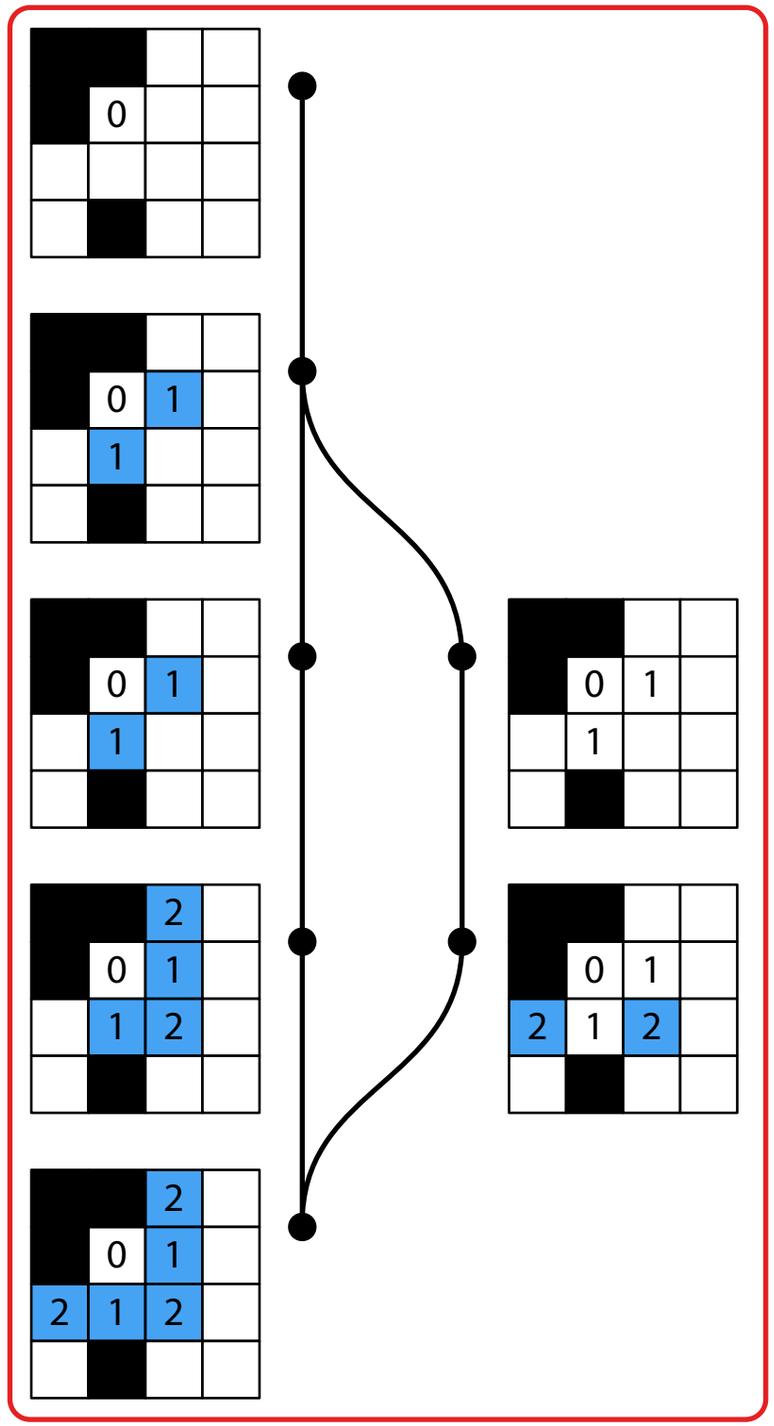

```
(atomic
  …
  (join child))
```

merge local store of child into parent

Conflict resolution function: `(ref 0 resolve)`

All tasks commit atomically

⇒ isolation and progress maintained



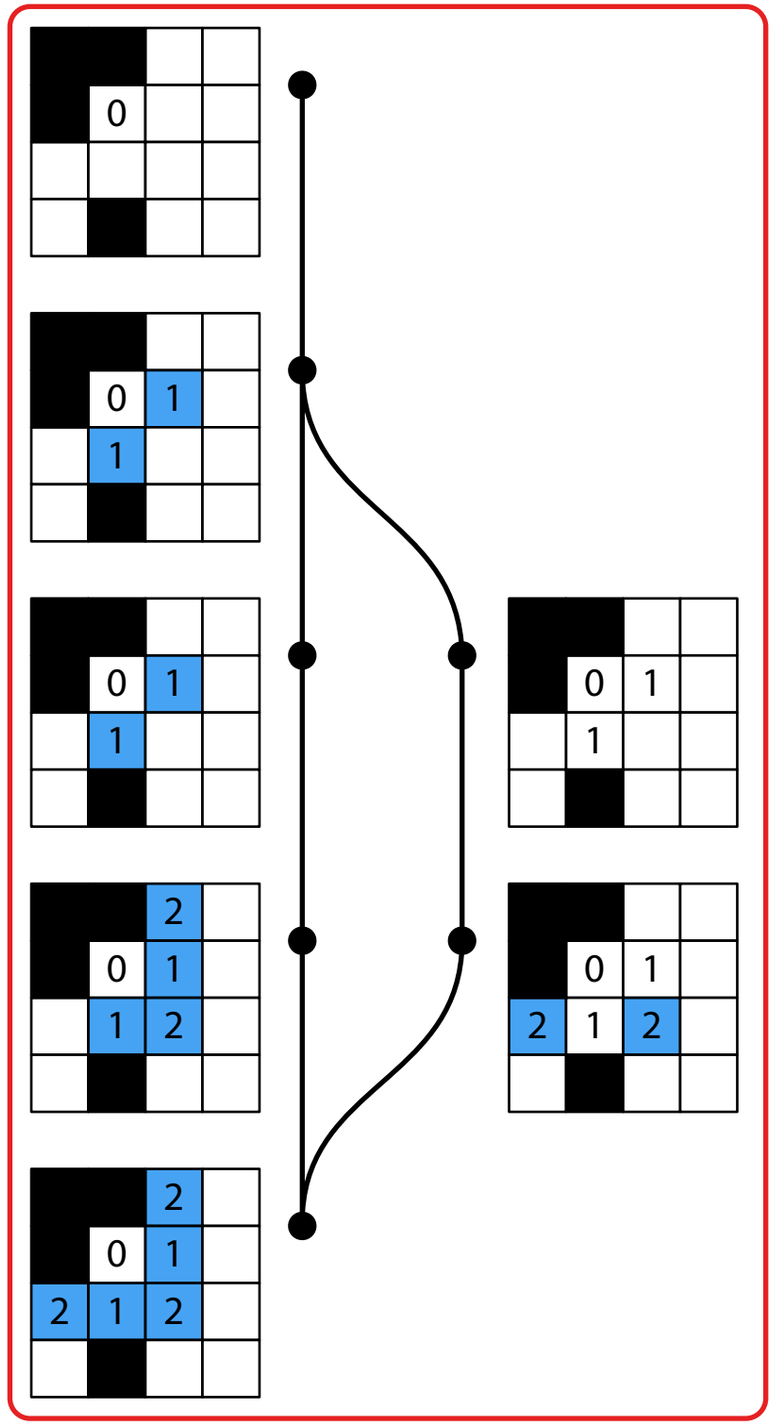

```
(atomic
  …
  (join child))
```

All tasks must be joined before commit

⇒ **isolation maintained** Iso

progress maintained Pro

Intratransaction determinacy



Transactions can commit in any order

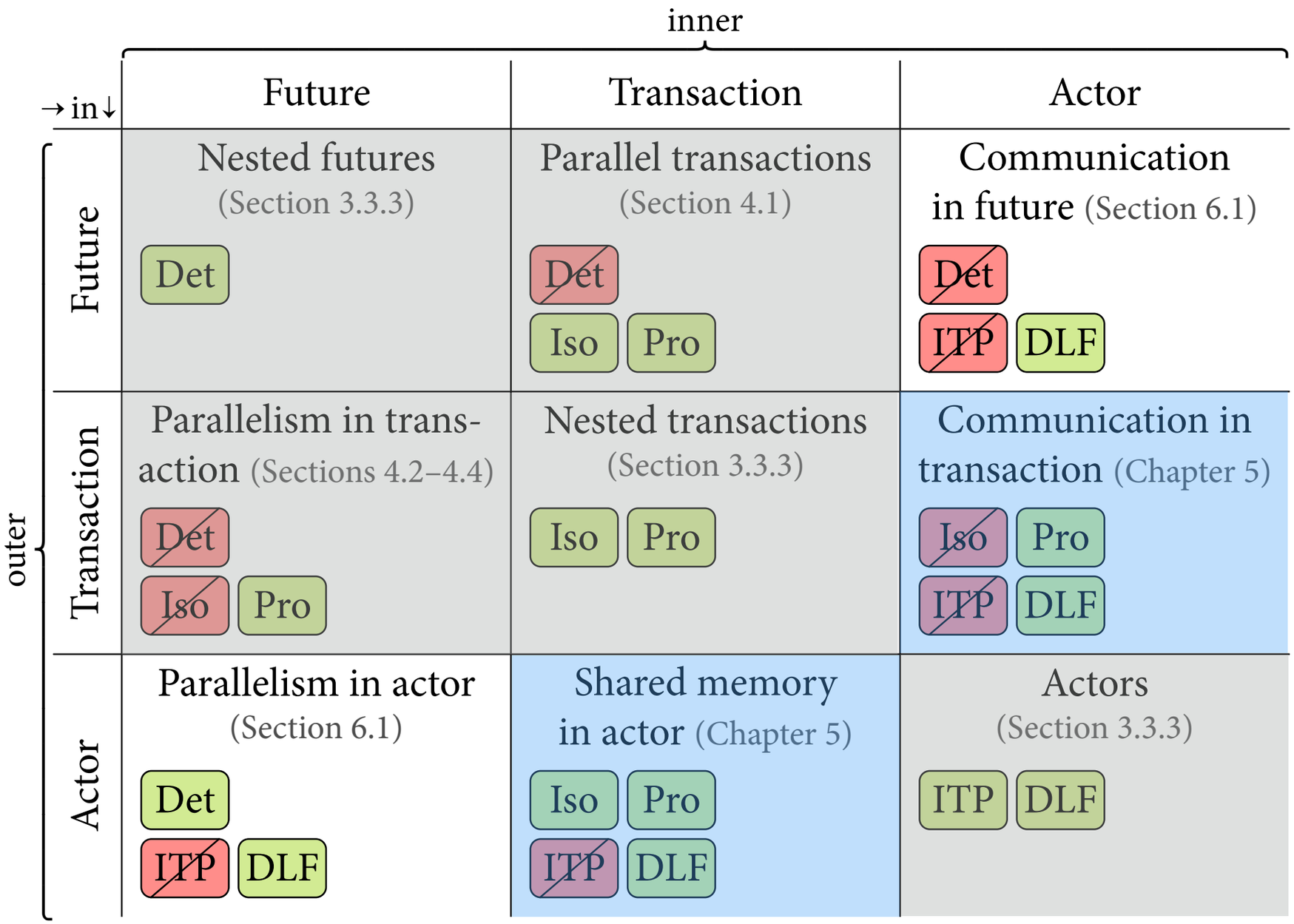
⇒ Det inevitable

But: determinacy *within* each transaction

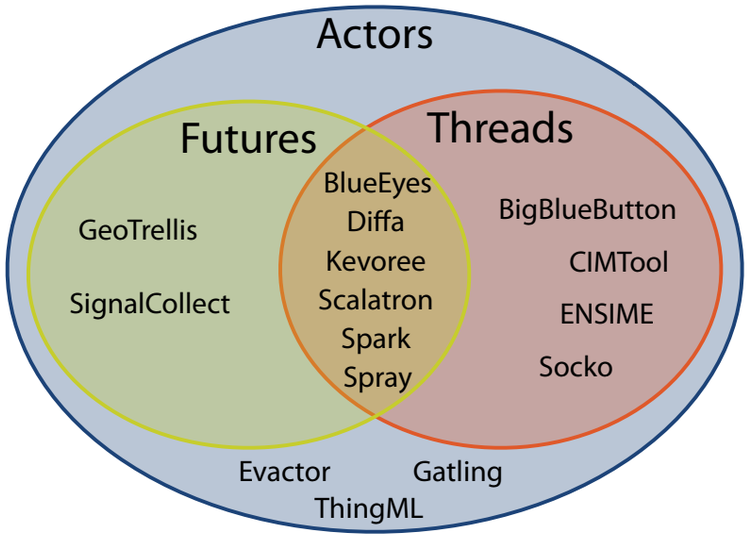
= **intratransaction determinacy** ITD

And isolation *between* transactions Iso

Transactions + Actors



Motivation: (1) safe shared information between actors

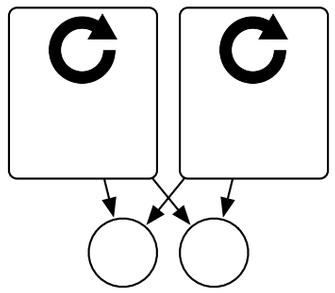


Impure actor languages (e.g. Scala)

10/15 projects introduce shared memory

⇒ **ITP broken** ITP

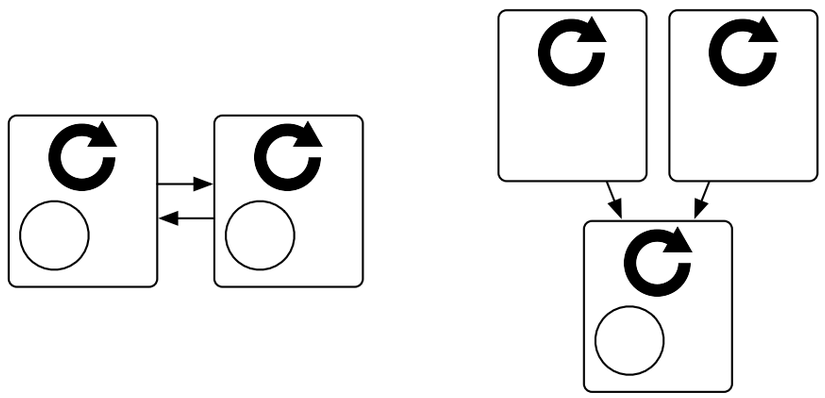
⇒ **races & deadlocks possible**



Pure actor languages (e.g. Erlang)

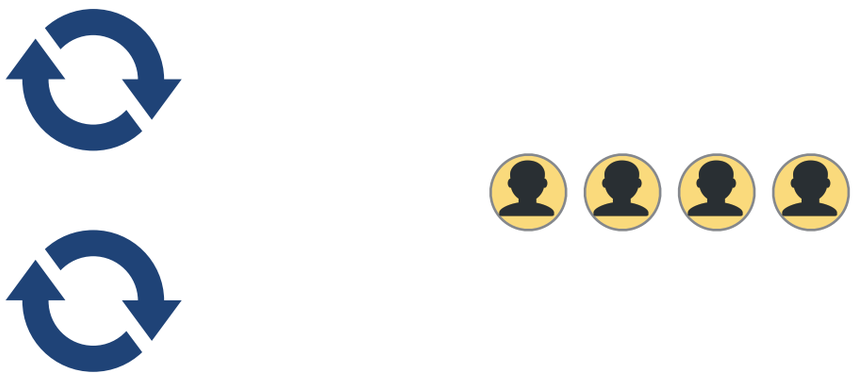
Patterns: replication/delegation

⇒ ITP guaranteed but
safety up to the developer



Motivation: (2) communication between transactions

"Vacation" processes customers in parallel

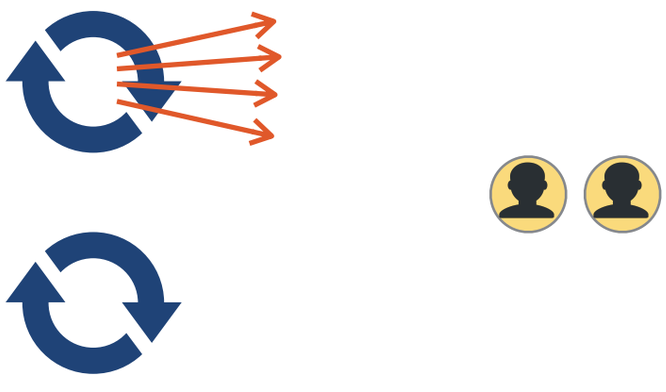


```
(def customer-behavior
  (behavior [id] [c]
    (atomic
      (reserve-flight (:orig @c) (:dest @c) (:start @c))
      (reserve-flight (:dest @c) (:orig @c) (:end @c))
      (reserve-room   (:dest @c) (:start @c) (:end @c))
      (reserve-car    (:dest @c) (:start @c) (:end @c))
      (ref-set c (assoc @c :password (generate-password))))))
```

Minh, Chung, Kozyrakis, Olukotun (2008). *STAMP: Stanford Transactional Applications for Multi-Processing* (IISWC'08)

Motivation: (2) communication between transactions

but more fine-grained parallelism is possible



```
(def customer-behavior
  (behavior [id] [c]
    (atomic
      (send (rand workers) :flight (:orig @c) …)
      (send (rand workers) :flight (:dest @c) …)
      (send (rand workers) :room   (:dest @c) …)
      (send (rand workers) :car    (:dest @c) …)
      (ref-set c (assoc @c :password (generate-password))))))
```

⇒ isolation broken Iso

Transactional Actors

Make side effects on actors part of transaction

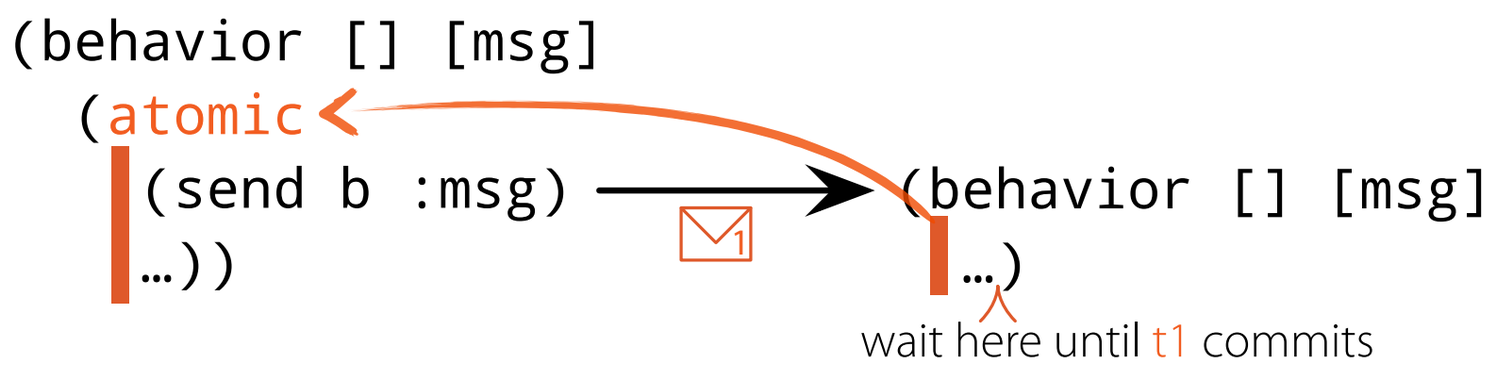
```
(atomic

  (def airline-beh
    (behavior [flights]
      …))

(spawn airline-beh flights)
(become airline-beh flights)

(send :process-customer
  (deref c)))
```

separate from transaction, no side-effect ✓

delay side effect until commit (pessimistic) ↷

sent immediately, but **rolled back** on abort (optimistic) ↶

Sending a message in a transaction

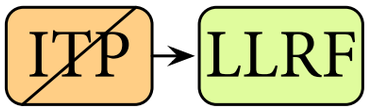


Message **depends** on the transaction

Receiving turn is **tentative**:

- Side effects (`spawn`, `become`) delayed
- Sends get dependency
- At the end, wait for dependency to commit

⇒ **isolation maintained** Iso
progress maintained Pro



Low-level Race Freedom

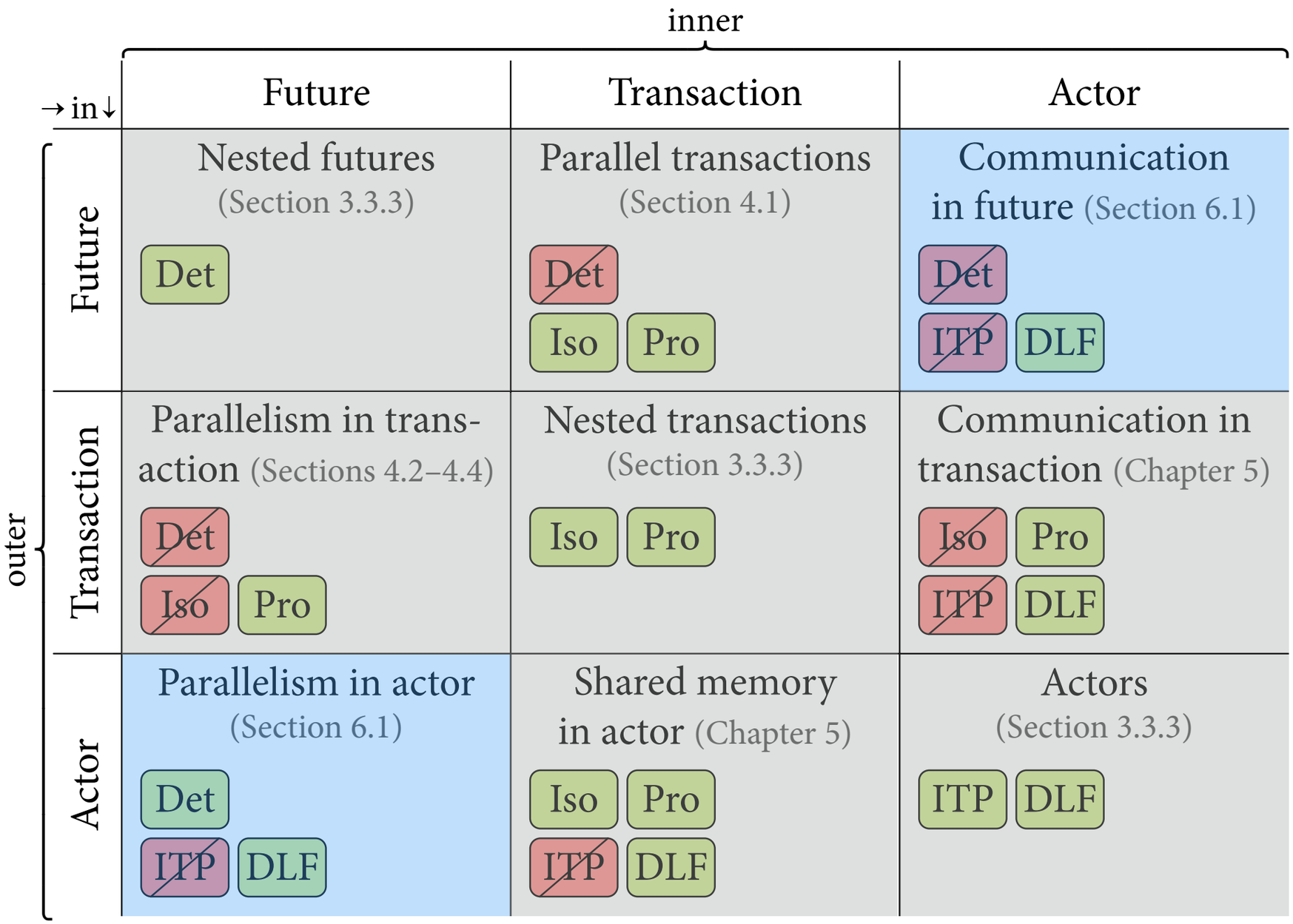
Shared memory ⇒ ITP broken ITP

But: **Low-level Race Freedom** LLRF

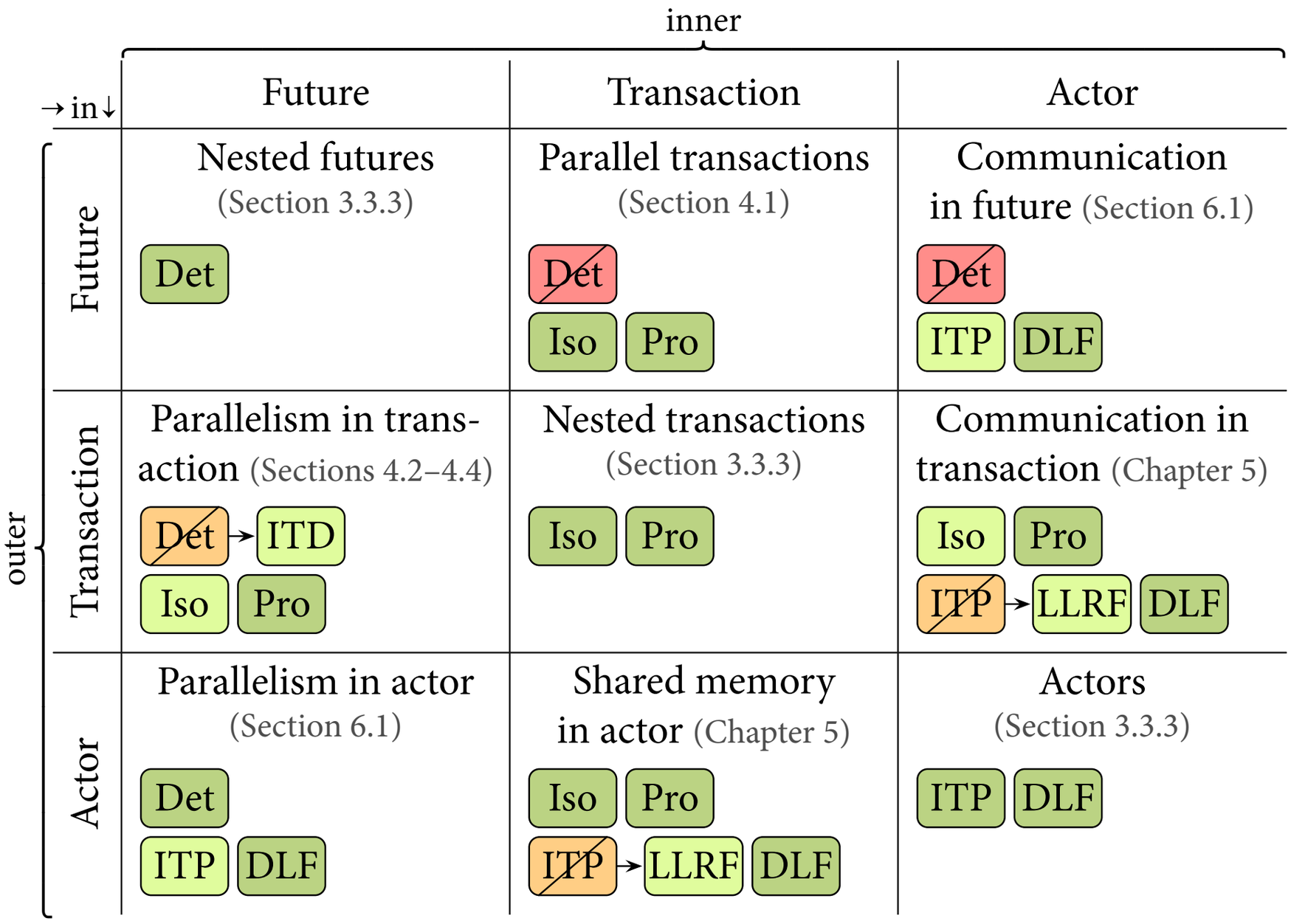
→ **shared memory** is isolated at level of **transactions**

→ **private memory of actors** is isolated at level of **turns**

Actors + Futures



Chocola:
c^h^omposable concurrency language



Implementation

Extension of Clojure

- **Futures & Transactions**: built into Clojure
- **Actors**: simple implementation
- Transactional Futures
- Transactional Actors

} added

☞ Chapter 8

☞ http://soft.vub.ac.be/~jswalens/chocola

Formalization of Operational Semantics “PureChocola”

Uniform formalization of three separate models ☞ Chapter 2

Program state	p	::=	$\langle \mathrm{T} \rangle$
Tasks	$\mathrm{T} \subset$ Task		
Task	task $\in$ Task	::=	$\langle f, e \rangle$

Program state	p	::=	$\langle \mathrm{T}, \tau, \sigma \rangle$
Task	task $\in$ Task	::=	$\langle f, e, \mathsf{n}^? \rangle$
Transactions	τ : TransactionNumber $\rightharpoonup$ Transaction		
snapshot, local store	$\sigma, \overleftarrow{\sigma}, \delta$: TVar $\rightharpoonup$ Value		
Transaction	tx $\in$ Transaction	::=	$\langle \circ, \overleftarrow{\sigma}, \overleftarrow{e}, \delta \rangle$
Transaction id	n $\in$ TransactionNumber		$= \mathbb{N}^+$
Transaction state	$\circ$	::=	$\triangleright \mid \checkmark \mid \times$

Program state	p	::=	$\langle \mathrm{A}, \mu \rangle$
Actors	$\mathrm{A} \subset$ Actor		
Inboxes	μ : Address $\rightharpoonup \overline{\text{Message}}$		
Actor	act $\in$ Actor	::=	$\langle a, e^?, \text{beh} \rangle$
Behavior	beh $\in$ Behavior	::=	$\langle b, \overline{v} \rangle$
Message	msg $\in$ Message	::=	$\langle a_{\text{from}}, a_{\text{to}}, \overline{v} \rangle$

Program state	p	::=	$\langle \mathrm{A}, \mathrm{T}, \mu, \tau, \sigma \rangle$
Actors	$\mathrm{A} \subset$ Actor		
Tasks	$\mathrm{T} \subset$ Task		
Inboxes	μ : Address $\rightharpoonup \overline{\text{Message}}$		
Transactions	τ : TransactionNumber $\rightharpoonup$ Transaction		
Transactional heap	σ : TVar $\rightharpoonup$ Value		
Actor	act $\in$ Actor	::=	$\langle a, f^?_{\text{root}}, \text{beh}, \mathsf{n}^?_{\text{dep}} \rangle$
Task	task $\in$ Task	::=	$\langle f, a, e, \mathrm{F_s}, \mathrm{F_j}, \text{eff}, \text{ctx}^? \rangle$
Transaction	tx $\in$ Transaction	::=	$\langle \circ, \overleftarrow{e} \rangle$
Spawned and joined futures	$\mathrm{F_s}, \mathrm{F_j} \subset$ Future		
Effects on actors	eff	::=	$\langle \vec{\mathrm{A}}, \overrightarrow{\text{beh}}^? \rangle$
Transactional context	ctx	::=	$\langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle$
Message	msg $\in$ Message	::=	$\langle a_{\text{from}}, a_{\text{to}}, \overline{v}, \mathsf{n}^?_{\text{dep}} \rangle$
As before:			
Behavior	beh $\in$ Behavior	::=	$\langle b, \overline{v} \rangle$
Snapshot, local store	$\overleftarrow{\sigma}, \delta$: TVar $\rightharpoonup$ Value		
Transaction id	n $\in$ TransactionNumber		
Transaction state	$\circ$	::=	$\triangleright \mid \checkmark \mid \times$

Formalization of Chocola ☞ Chapter 7

Formalization

same constructs, but take context into account

$\underline{\text{commit}_{\checkmark}|_{\text{c}}}$

$$\langle \text{A} \uplus \mathsf{act}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathtt{atomic}\star\, v], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \rangle \rangle, \mu, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{e} \rangle], \sigma \rangle$$
$$\to \langle \text{A} \uplus \mathsf{act}, \text{T} \uplus \langle f, a, \mathcal{E}[v], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}_+, \bullet \rangle, \mu, \tau[\mathsf{n} \mapsto \langle \checkmark, \overleftarrow{e} \rangle], \sigma :: \delta \rangle$$
$$\text{where } \mathsf{act} = \langle a, f_{\text{root}}, \mathsf{beh}, \mathsf{n}^?_{\text{dep}} \rangle$$
$$\text{if } \forall r \in \text{dom}(\delta) : \sigma(r) = \overleftarrow{\sigma}(r) \qquad \text{(no conflicts)}$$
$$\forall f_\star \in \text{tx-futs}(\text{T}, \mathsf{n}) : f_\star \in \text{F}_\text{j} \qquad \text{(all futures spawned in the tx must have been joined)}$$
$$\mathsf{n}^?_{\text{dep}} = \bullet \text{ or } \tau(\mathsf{n}^?_{\text{dep}}) = \langle \checkmark, \overleftarrow{e} \rangle \qquad \text{(in a definitive or a success}$$
$$\text{with } \mathsf{eff}_+ = \mathsf{eff} \mathrel{+}= \mathsf{eff}_{\text{tx}}$$

$\underline{\text{commit}_{\text{✗}}|_{\text{c}}}$

$$\langle \text{A}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathtt{atomic}\star\, v], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \rangle \rangle, \mu, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{e} \rangle], \sigma$$
$$\to \langle \text{A}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathtt{atomic}\ \overleftarrow{e}], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}, \bullet \rangle, \mu, \tau[\mathsf{n} \mapsto \langle \text{✗}, \overleftarrow{e} \rangle], \sigma \rangle$$
$$\text{if } \exists r \in \text{dom}(\delta) : \sigma(r) \neq \overleftarrow{\sigma}(r) \qquad \text{(a c}$$
$$\forall f_\star \in \text{tx-futs}(\text{T}, \mathsf{n}) : f_\star \in \text{F}_\text{j} \qquad \text{(all futures spawned in the tx must h}$$

$\underline{\text{commit}_{\bullet}|_{\text{c}}}$

$$\langle \text{A} \uplus \mathsf{act}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathtt{atomic}\star\, v], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}, \mathsf{ctx} \rangle, \mu, \tau, \sigma \rangle$$
$$\to \langle \text{A} \uplus \mathsf{act}', \text{T}', \mu, \tau', \sigma \rangle$$
$$\text{where } \mathsf{act} = \langle a, f_{\text{root}}, \mathsf{beh}, \mathsf{n}_{\text{dep}} \rangle$$
$$\text{if } \tau(\mathsf{n}_{\text{dep}}) = \langle \text{✗}, \overleftarrow{e} \rangle \qquad \text{(in a fail}$$
$$\text{with } \mathsf{act}' = \langle a, \bullet, \mathsf{beh}, \bullet \rangle \qquad \text{(reset a}$$
$$\text{T}' = \text{T} \setminus \text{actor-tasks}(\text{T}, a) \qquad \text{(abort and remove all t}$$
$$\tau'(\mathsf{n}) = \begin{cases} \langle \text{✗}, \mathtt{nil} \rangle & \text{if } \mathsf{n} \in \text{actor-txs}(a) \qquad \text{(abort all tr} \\ \tau(\mathsf{n}) & \text{otherwise} \qquad \text{turn, i} \end{cases}$$

$\underline{\text{spawn}|_{\text{c}}}$

$$\langle \text{A}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathsf{spawn}\ b_\star\ \overline{v}], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}, \mathsf{ctx}^? \rangle, \mu, \tau, \sigma \rangle$$
$$\to \langle \text{A}, \text{T} \uplus \langle f, a, \mathcal{E}[a_\star], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}', \mathsf{ctx}' \rangle, \mu[a_\star \mapsto []], \tau, \sigma \rangle$$
$$\text{with } a_\star \text{ fresh}$$
$$\mathsf{act}_\star = \langle a_\star, \bullet, \langle b_\star, \overline{v} \rangle, \bullet \rangle$$
$$\begin{cases} \text{if } \mathsf{ctx}^? = \bullet\text{:} & \mathsf{ctx}' = \bullet \qquad \text{(outside transaction)} \\ & \mathsf{eff}' = \mathsf{eff} \mathrel{+}= \langle \mathsf{act}_\star, \bullet \rangle \\ \text{if } \mathsf{ctx}^? = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \rangle\text{:} & \mathsf{ctx}' = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \mathrel{+}= \langle \mathsf{act}_\star, \bullet \rangle \rangle \qquad \text{(in transaction)} \\ & \mathsf{eff}' = \mathsf{eff} \end{cases}$$

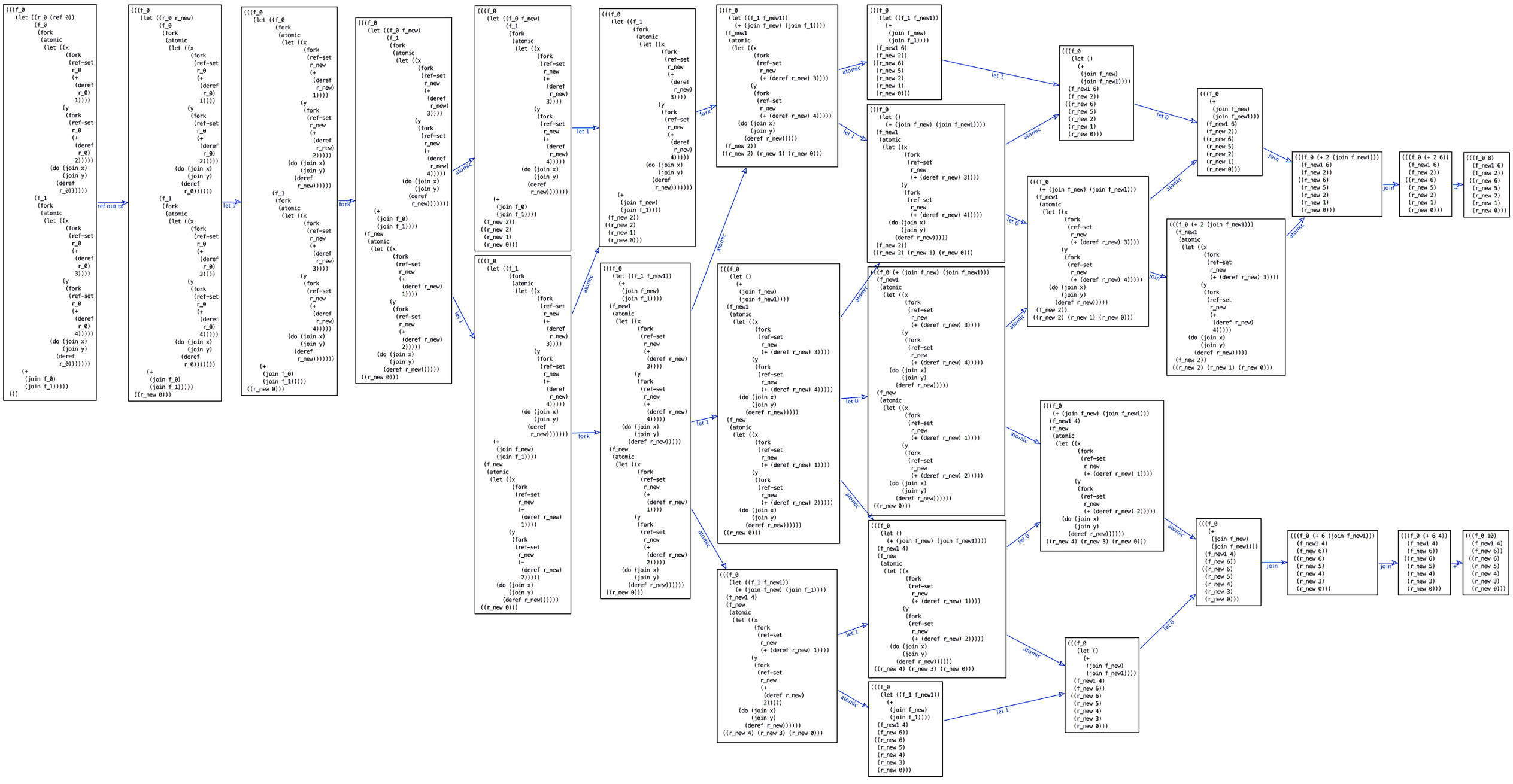
$\underline{\text{become}|_{\text{c}}}$

$$\langle \text{A}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathsf{become}\ b_\star\ \overline{v}], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}, \mathsf{ctx}^? \rangle, \mu, \tau, \sigma \rangle$$
$$\to \langle \text{A}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathtt{nil}], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}', \mathsf{ctx}' \rangle, \mu, \tau, \sigma \rangle$$
$$\text{with } \begin{cases} \text{if } \mathsf{ctx}^? = \bullet\text{:} & \mathsf{ctx}' = \bullet \qquad \text{(outside transaction)} \\ & \mathsf{eff}' = \mathsf{eff} \mathrel{+}= \langle \varnothing, \langle b_\star, \overline{v} \rangle \rangle \\ \text{if } \mathsf{ctx}^? = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \rangle\text{:} & \mathsf{ctx}' = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \mathrel{+}= \langle \varnothing, \langle b_\star, \overline{v} \rangle \rangle \rangle \qquad \text{(in transaction)} \\ & \mathsf{eff}' = \mathsf{eff} \end{cases}$$

$\underline{\text{send}|_{\text{c}}}$

$$\langle \text{A} \uplus \mathsf{act}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathsf{send}\ a_{\text{to}}\ \overline{v}], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}, \mathsf{ctx}^? \rangle, \mu[a_{\text{to}} \mapsto \overline{\mathsf{msg}}], \tau, \sigma \rangle$$
$$\to \langle \text{A} \uplus \mathsf{act}, \text{T} \uplus \langle f, a, \mathcal{E}[\mathtt{nil}], \text{F}_\text{s}, \text{F}_\text{j}, \mathsf{eff}', \mathsf{ctx}' \rangle, \mu[a_{\text{to}} \mapsto \overline{\mathsf{msg}} \cdot \mathsf{msg}], \tau, \sigma \rangle$$
$$\text{where } \mathsf{act} = \langle a, f_{\text{root}}, \mathsf{beh}, \mathsf{n}^?_{\text{dep}} \rangle$$
$$\text{with } \mathsf{msg} = \langle a, a_{\text{to}}, \overline{v}, \mathsf{n}^?_{\text{msg}} \rangle$$
$$\mathsf{n}^?_{\text{msg}} = \begin{cases} \mathsf{n}_{\text{tx}} & \text{if } \mathsf{ctx}^? = \langle \mathsf{n}_{\text{tx}}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \rangle \qquad \text{(in transaction)} \\ \mathsf{n}^?_{\text{dep}} & \text{if } \mathsf{ctx}^? = \bullet \text{ and } \mathsf{n}^?_{\text{dep}} \neq \bullet \qquad \text{(in tentative turn)} \\ \bullet & \text{otherwise} \qquad \text{(definitive)} \end{cases}$$

Executable formal semantics with PLT Redex



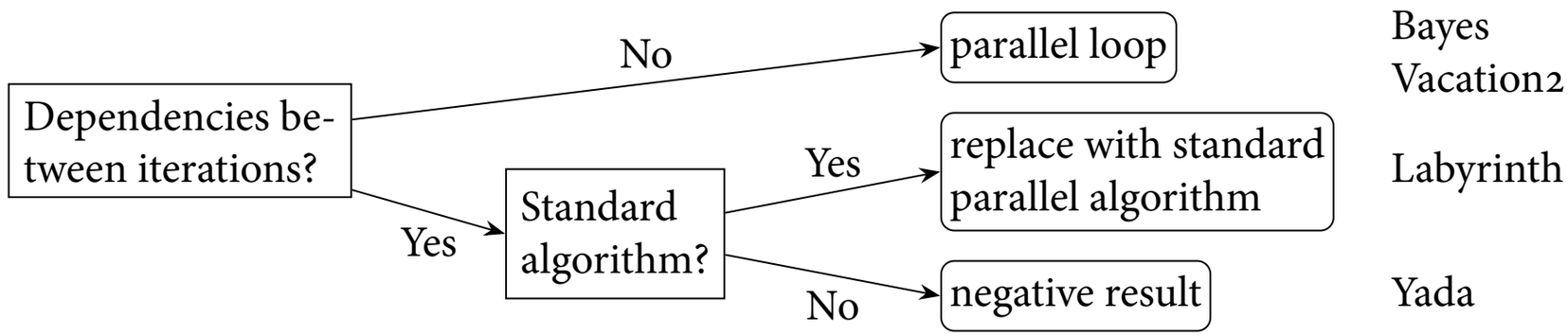
☞ https://github.com/jswalens/chocola-redex

Evaluation approach

① selection of benchmarks

Application	Transaction length (mean # of instructions per tx)	Average time in transaction
Labyrinth	219,571	100%
Bayes	60,584	83%
Yada	9,795	100%
Vacation-high	3,223	86%
Genome	1,717	97%
Intruder	330	33%
Kmeans-high	117	7%
SSCA2	50	17%

② parallelization

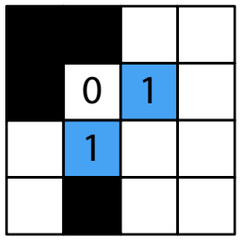


③ evaluation criteria

performance: speed-up

developer effort: lines changed + qualitative assessment

☞ https://github.com/jswalens/{labyrinth,bayes,yada,vacation2}



Labyrinth

Original:

```
(atomic
  (breadth-first-search …))
```

+11% LoC
-4% LoC

Transformed:

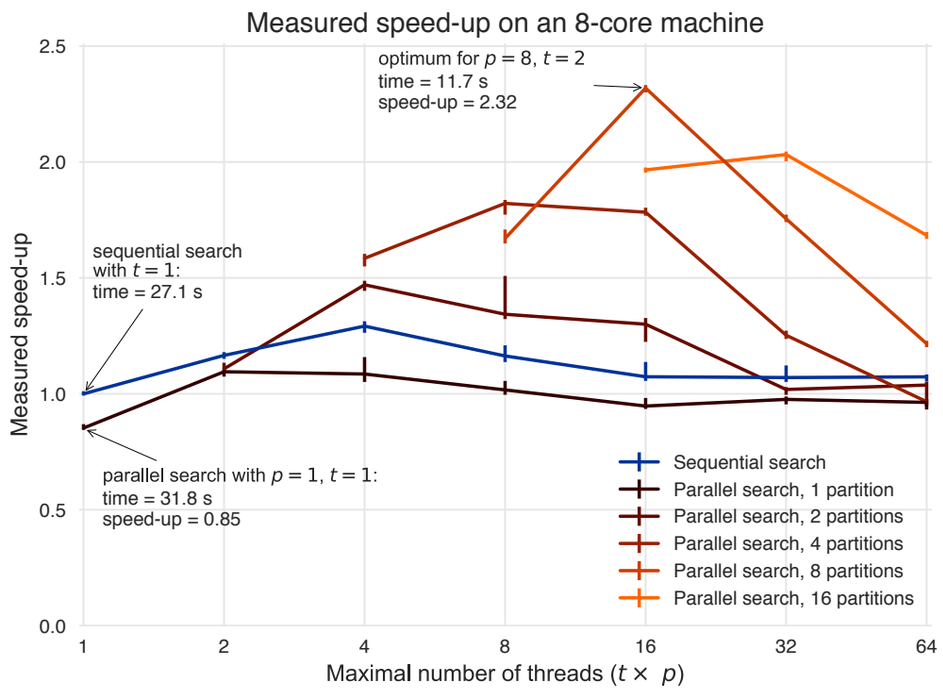
```
(atomic
  (parallel-bfs …))

      (defn parallel-bfs […]
        (for […]
          (fork …)))
```

Maximal speed-up = 1.3

~40% retries

Maximal speed-up = 2.3

~10% retries



Bayes

Original:

```
(atomic
  …
  (for [id (range (:n-var tree))]
    (compute-local-log-likelihood …)))
```

+1 LoC
-1 LoC

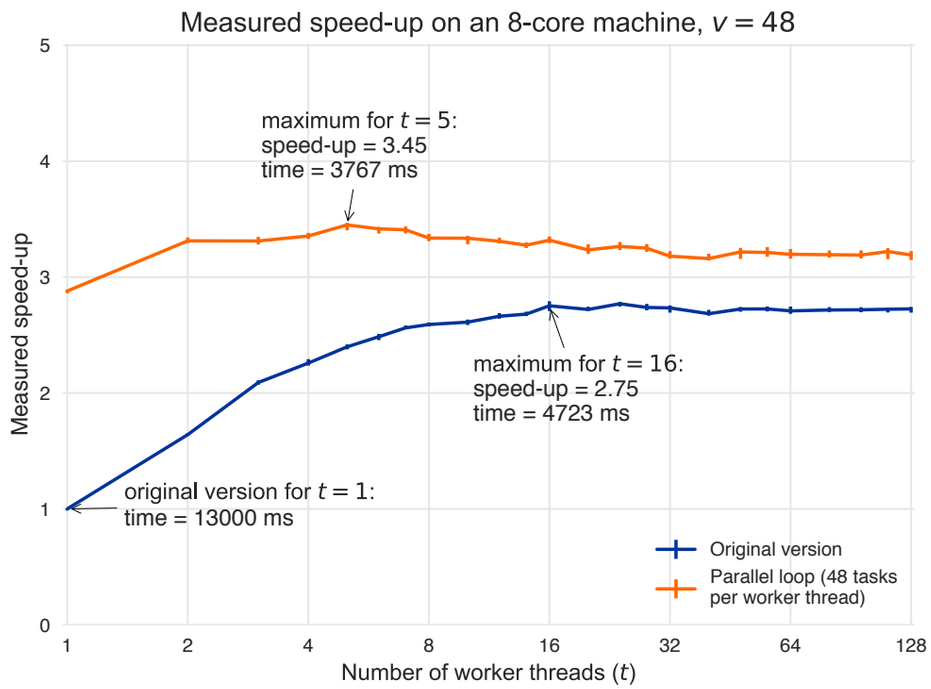
Transformed:

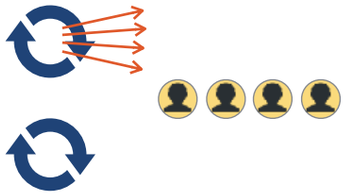
```
(atomic
  …
  (parallel-for [id (range …)]
    (compute-local-log-likelihood …)))
```

Maximal speed-up = 2.8

Maximal speed-up = 3.5

More fine-grained parallelism





Vacation2

Original:

```
(def customer-behavior
  (behavior [id] [c]
    (atomic
      (reserve-flight (:orig @c) (:dest @c) …)
      (reserve-flight (:dest @c) (:orig @c) …)
      (reserve-room   (:dest @c) (:start @c) …)
      (reserve-car    (:dest @c) (:start @c) …)
      (ref-set c (assoc @c :password …)))))
```

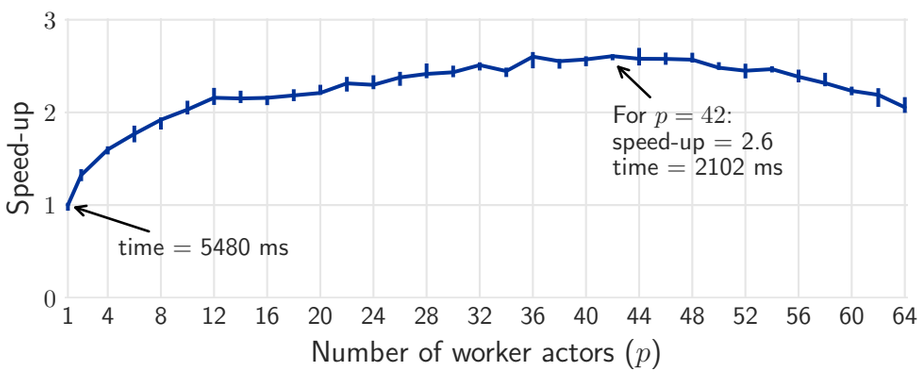
+8% LoC
-5% LoC

Transformed:

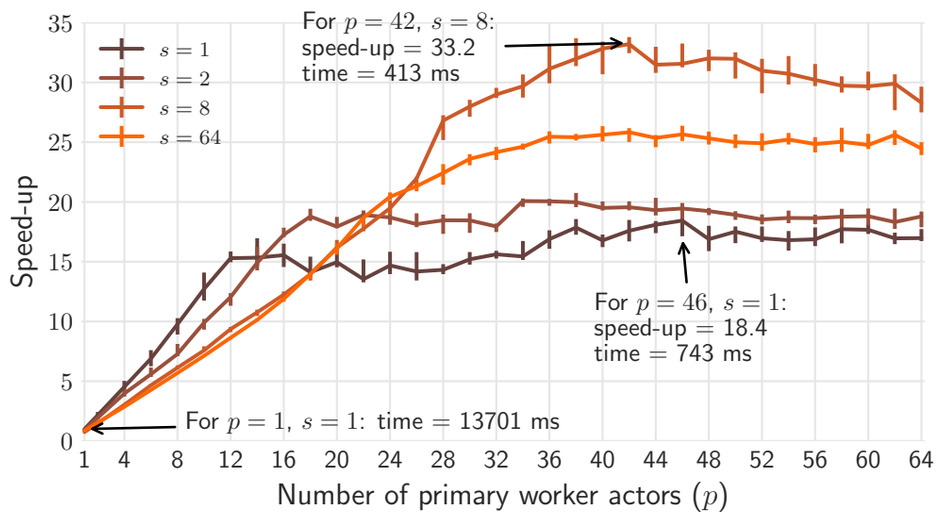
```
(def customer-behavior
  (behavior [id] [c]
    (atomic
      (send (rand workers) :flight (:orig @c) …)
      (send (rand workers) :flight (:dest @c) …)
      (send (rand workers) :room   (:dest @c) …)
      (send (rand workers) :car    (:dest @c) …)
      (ref-set c (assoc @c :password …)))))
```

Maximal speed-up = 2.6

Maximal speed-up = 33.2



More small transactions
⇒ fewer & cheaper conflicts



Contributions

- Systematic study of combinations of concurrency models in Clojure [Swalens et al., 2014]
- Systematic study of combinations of futures, transactions, and actors
- Transactional futures [Swalens et al., 2016]
- Transactional actors [Swalens et al., 2017]
- Unified framework – Chocola: [Swalens et al., 2018; submitted]
  - Implementation
  - Formal semantics
  - Evaluation

Swalens, Marr, De Koster, Van Cutsem (2014). *Towards Composable Concurrency Abstractions* (PLACES'14)
Swalens, De Koster, De Meuter (2016). *Transactional Tasks: Parallelism in Software Transactions* (ECOOP'16)
Swalens, De Koster, De Meuter (2017). *Transactional Actors: Communication in Transactions* (SEPS'17)
Swalens, De Koster, De Meuter (2018). *Chocola: Integrating Futures, Actors, and Transactions* (submitted to AGERE'18)

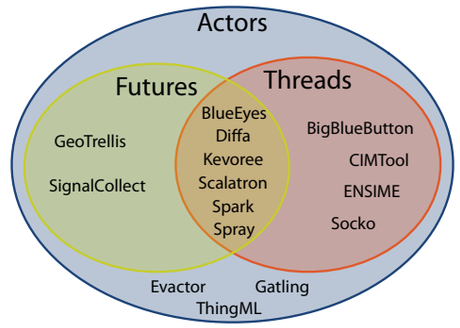
Future work

- Formal proofs of guarantees
- Other concurrency models
- Applicability & more benchmarks
- Comparison of implementation techniques

Conclusion

Concurrency models are combined

Naive combinations violate guarantees



outer \ inner	Atom	Agent	STM	Future	Promise	Channel
Races						
Atom's `swap!`	✗	✗	✗	✗	✗	✗
Agent's action	✓	✓	✓	✓	✓	✓
STM's `dosync`	✗	✓	✓	✗	✗	✗
Future	✓	✓	✓	✓	✓	✓
CSP's `go`	✓	✓	✓	✓	✓	✓

We studied the combinations of futures, transactions, and actors

→ Transactional Futures

→ Transactional Actors

↪ Chocola

→in↓	Future	Transaction	Actor
Future	Nested futures (Section 3.3.3) Det	Parallel transactions (Section 4.1) ~~Det~~ Iso Pro	Communication in future (Section 6.1) ~~Det~~ ITP DLF
Transaction	Parallelism in transaction (Sections 4.2–4.4) ~~Det~~ → ITD Iso Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) Iso Pro ~~ITP~~ → LLRF DLF
Actor	Parallelism in actor (Section 6.1) Det ITP DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ → LLRF DLF	Actors (Section 3.3.3) ITP DLF