A Multi-Paradigm Concurrent Programming Model

Janwillem Swalens

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

Software Languages Lab

VUB
VRIJE
UNIVERSITEIT
BRUSSEL

Q: Why are there so many programming languages?

Different tools for different jobs

strict ↔ flexible

✈ f

general purpose ↔ domain specific

🌐 X

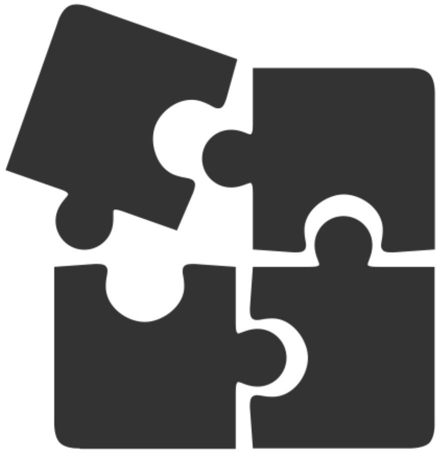
fast programs ↔ fast development

🎮 UBER eats

Q: Why create a new programming language?

Research technique

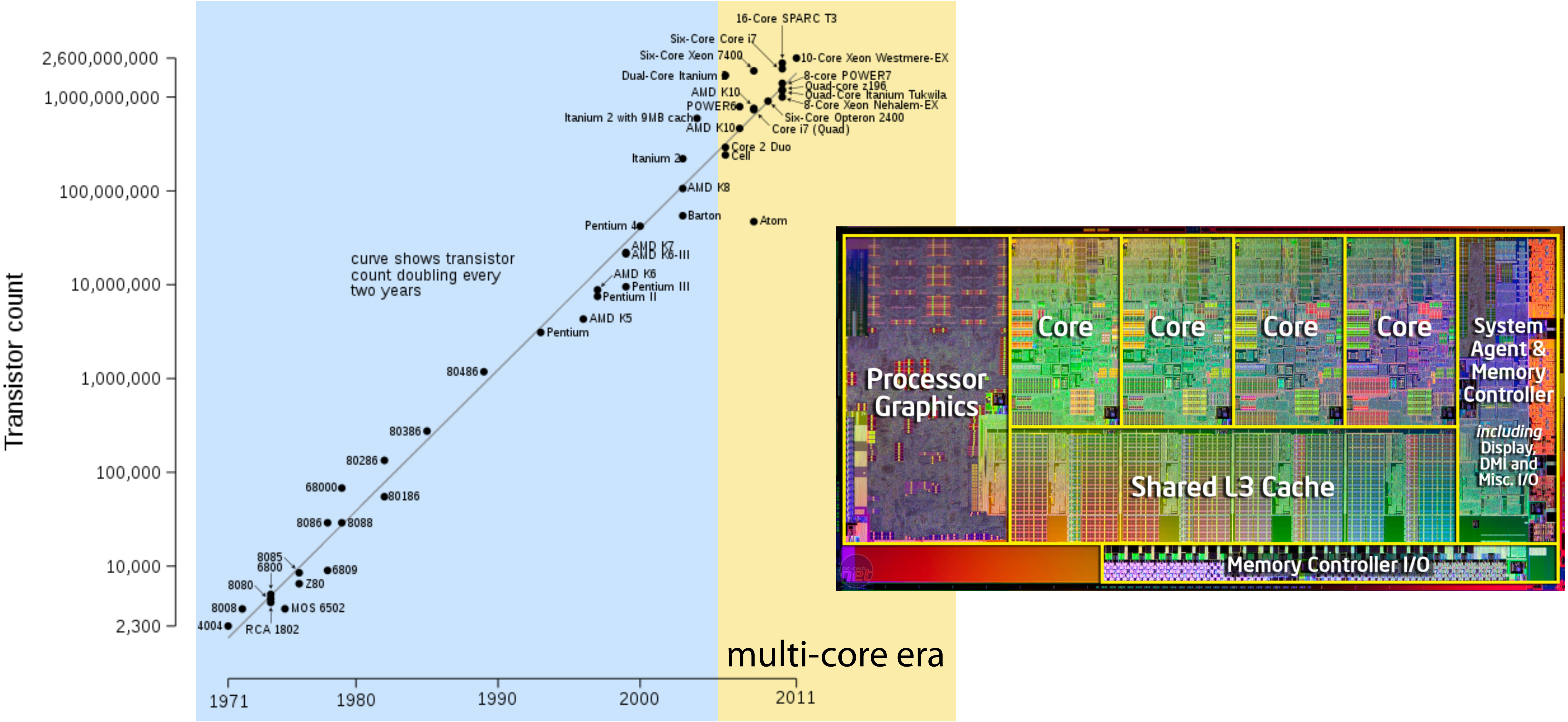
Small language with features we want to study



Can later be added to existing programming languages

Multi-core processors

Moore's law:
# transistors on chip doubles every two years



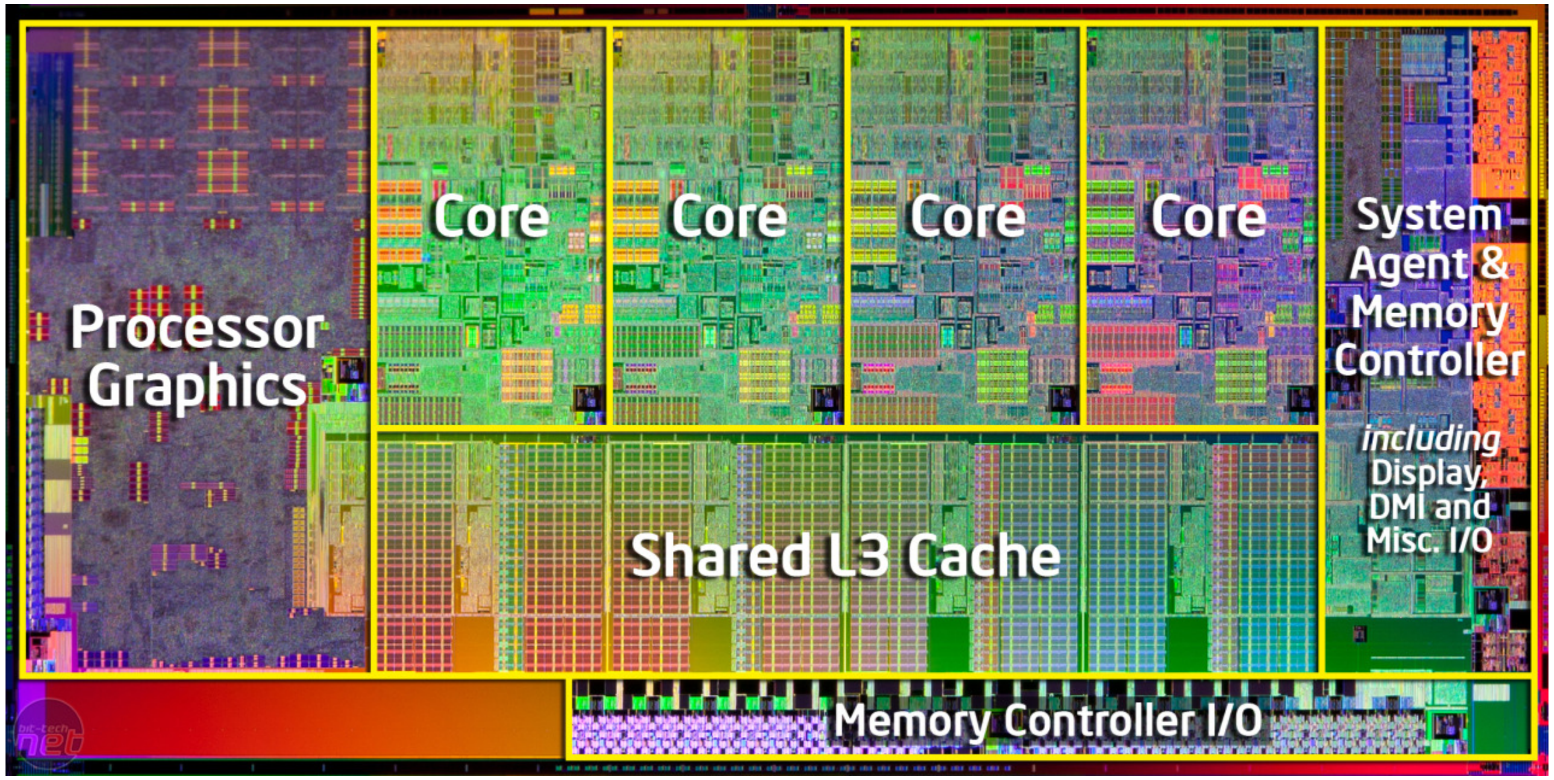
multi-core era

processors become faster because they execute instructions faster

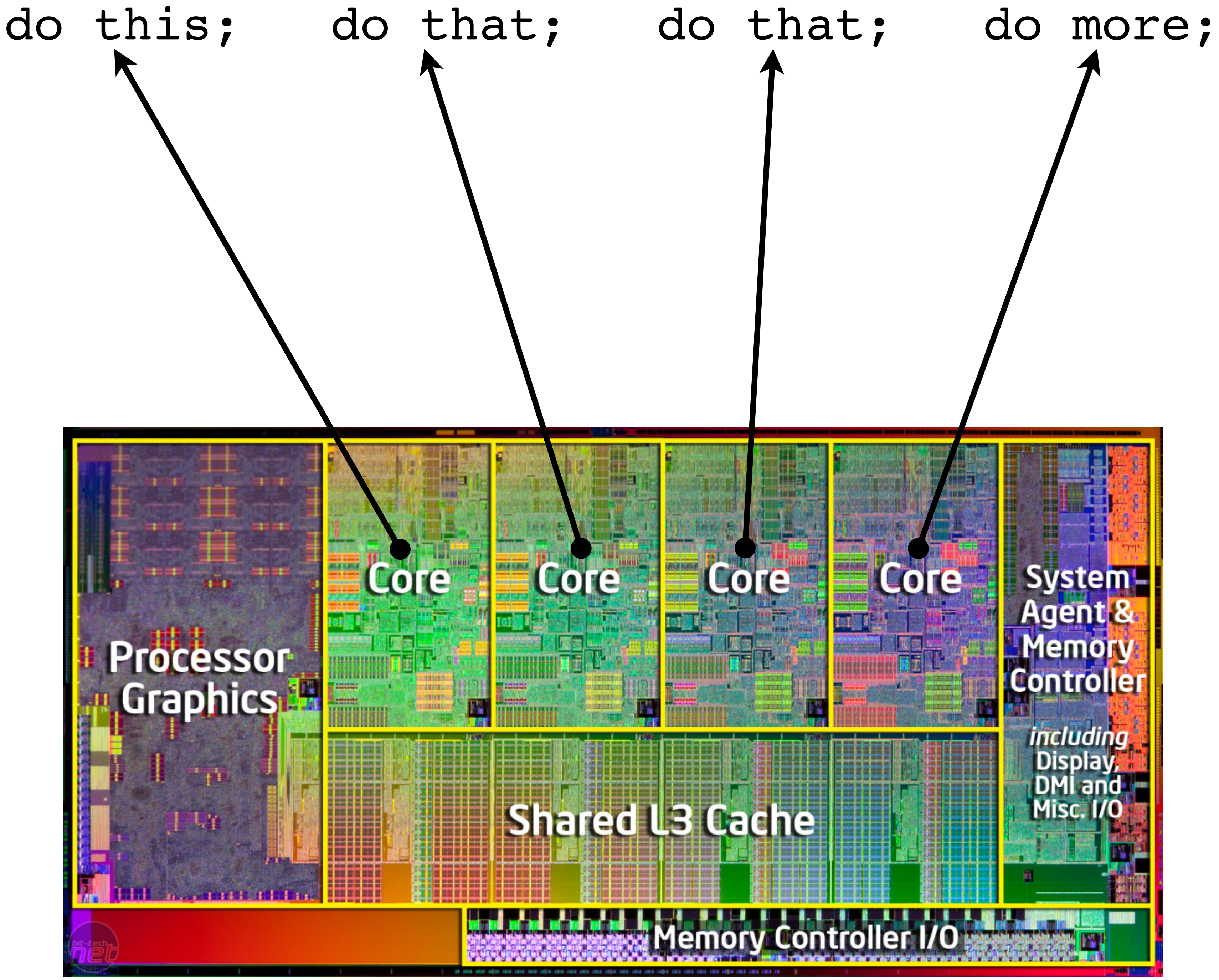
processors become faster because they execute more instructions in parallel

Sequential program

```
do this;
do that;
do that;
do more;
```

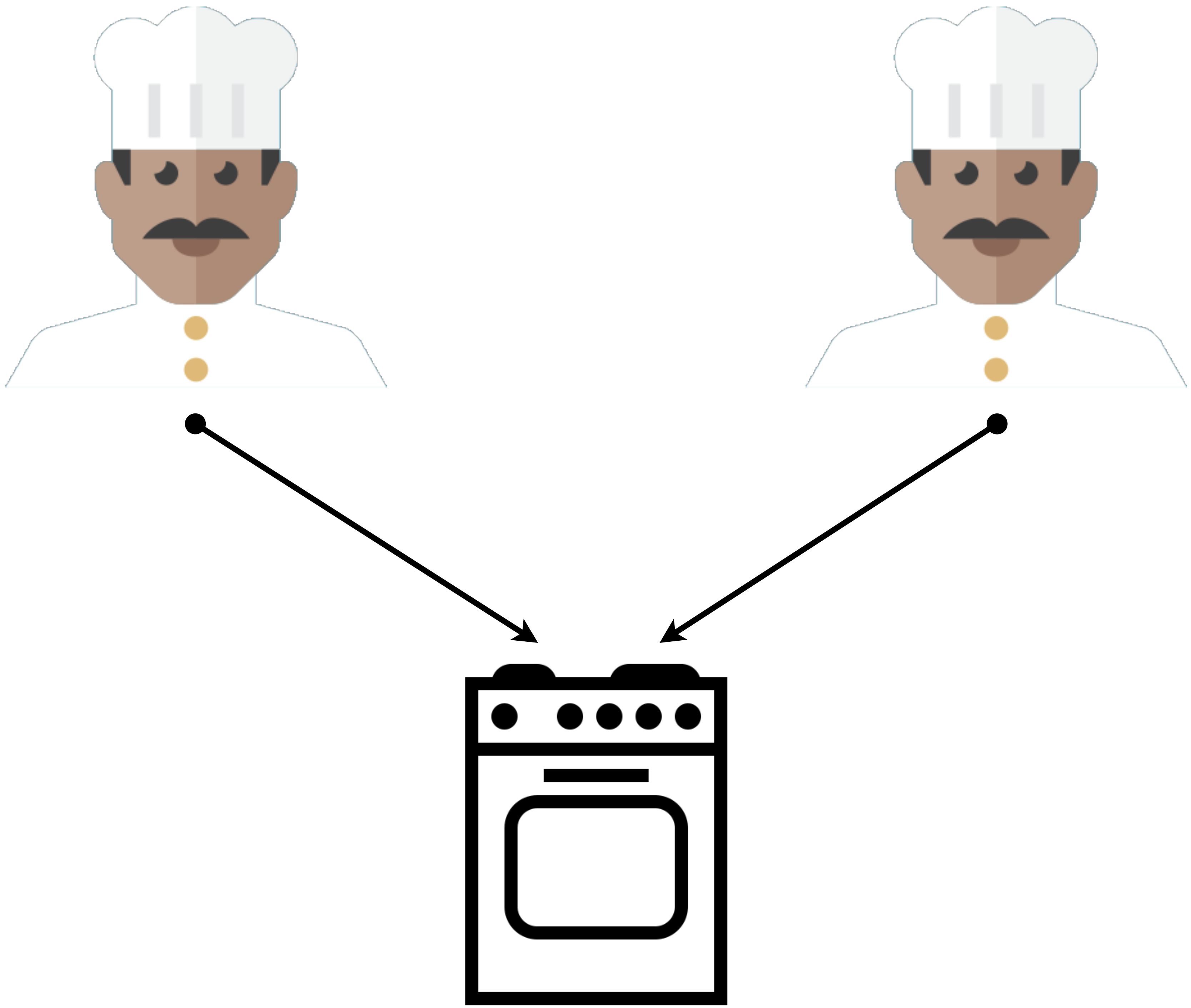


Program with concurrency

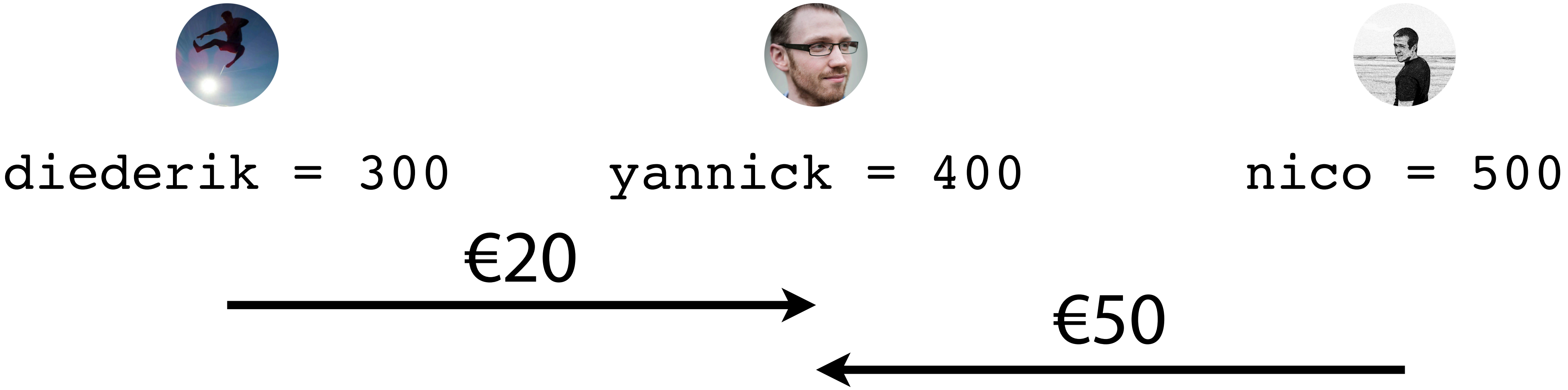


`do this;`   `do that;`   `do that;`   `do more;`

Concurrency is difficult



Concurrency is difficult



```
diederik = 300      yannick = 400      nico = 500
```

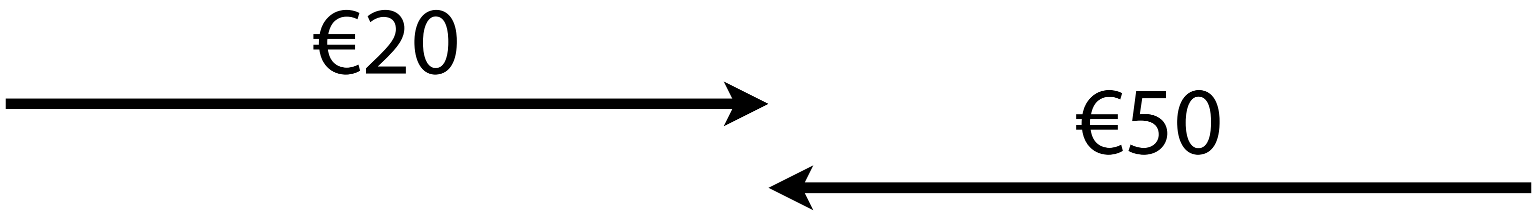
```
def transfer(a, b, amount):
  a = a - amount
  b = b + amount
```

Concurrency is difficult

`diederik = 300`

`yannick = 400`

`nico = 500`



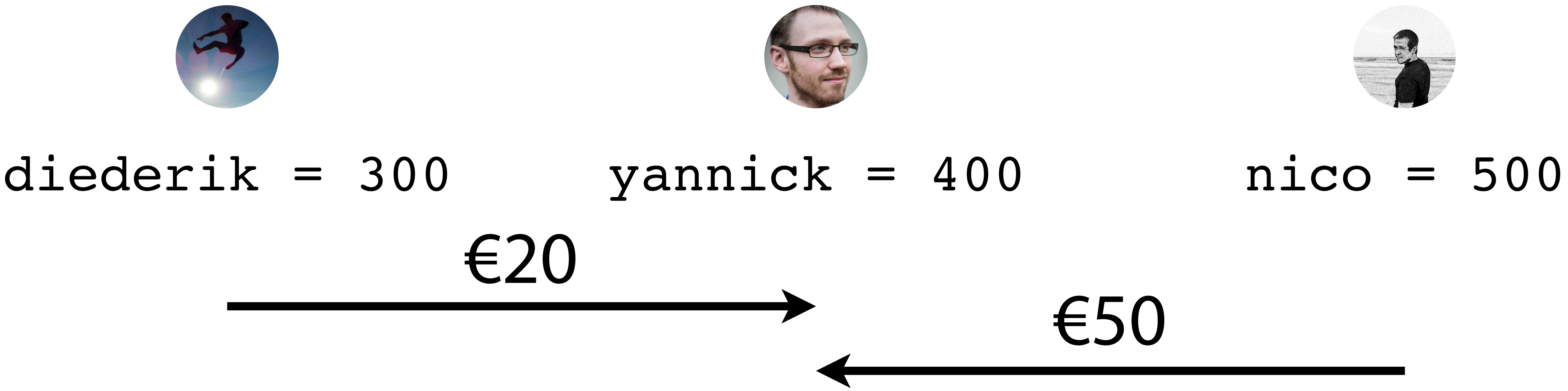
€20

€50

```
diederik = diederik - 20
yannick = yannick + 20
```

```
nico = nico - 50
yannick = yannick + 50
```

Concurrency is difficult



```
diederik = 300      yannick = 400      nico = 500
```

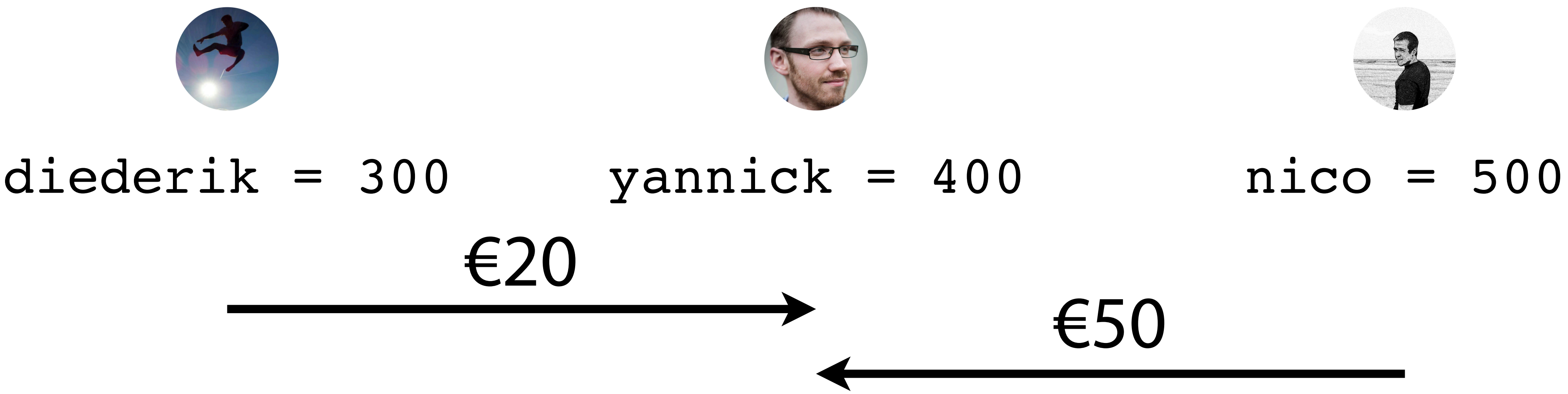
€20

€50

```
diederik = 300 - 20
yannick = yannick + 20
```

```
nico = nico - 50
yannick = yannick + 50
```

Concurrency is difficult



```
diederik = 300      yannick = 400      nico = 500
```

€20

€50

```
diederik = 300 - 20
yannick = yannick + 20
```

```
nico = 500 - 50
yannick = yannick + 50
```

Concurrency is difficult

```
diederik = 300      yannick = 400      nico = 500
```

€20

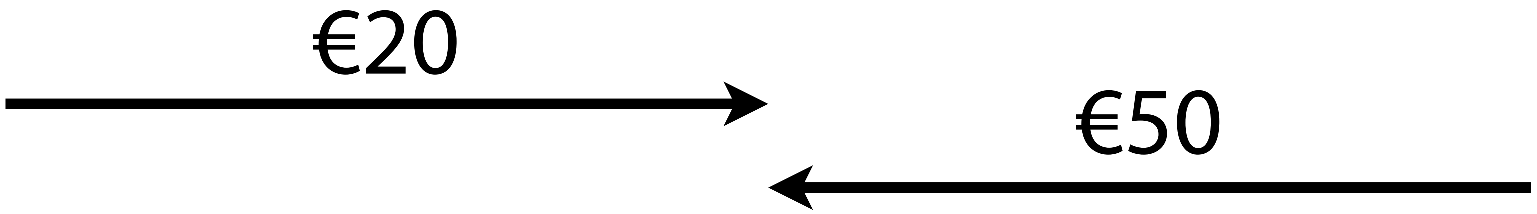
€50

```
diederik = 280
yannick = yannick + 20
```

```
nico = 450
yannick = yannick + 50
```

Concurrency is difficult

```
diederik = 280     yannick = 400     nico = 500
```

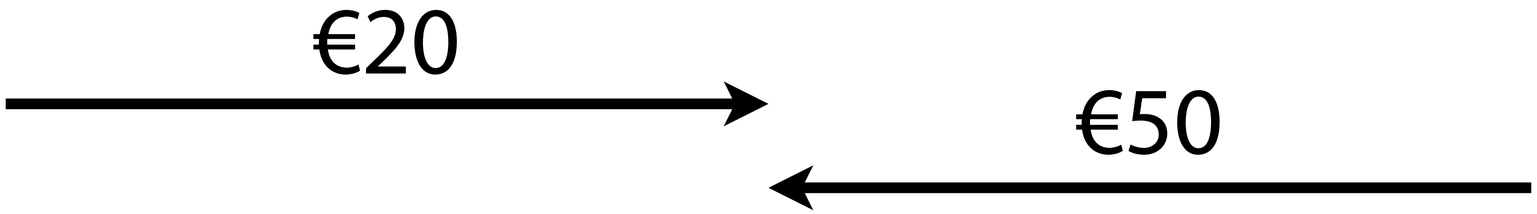


```
diederik = 280
yannick = yannick + 20
```

```
nico = 450
yannick = yannick + 50
```

Concurrency is difficult

```
diederik = 280      yannick = 400      nico = 450
```



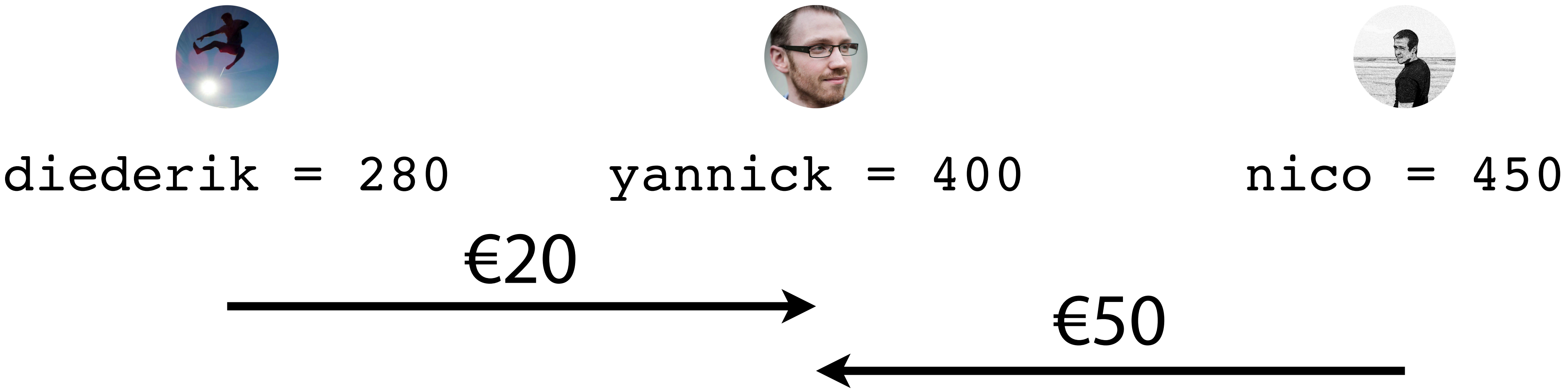
€20

€50

```
diederik = 280
yannick = yannick + 20
```

```
nico = 450
yannick = yannick + 50
```

Concurrency is difficult

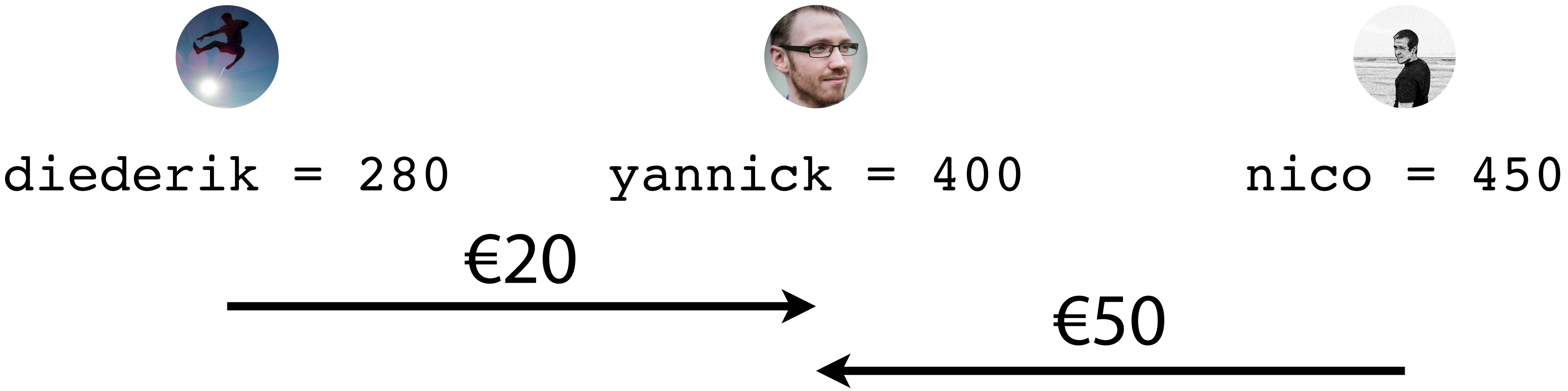


```
diederik = 280      yannick = 400      nico = 450
```

```
diederik = 280
yannick = 400 + 20
```

```
nico = 450
yannick = yannick + 50
```

Concurrency is difficult



diederik = 280

yannick = 400

nico = 450

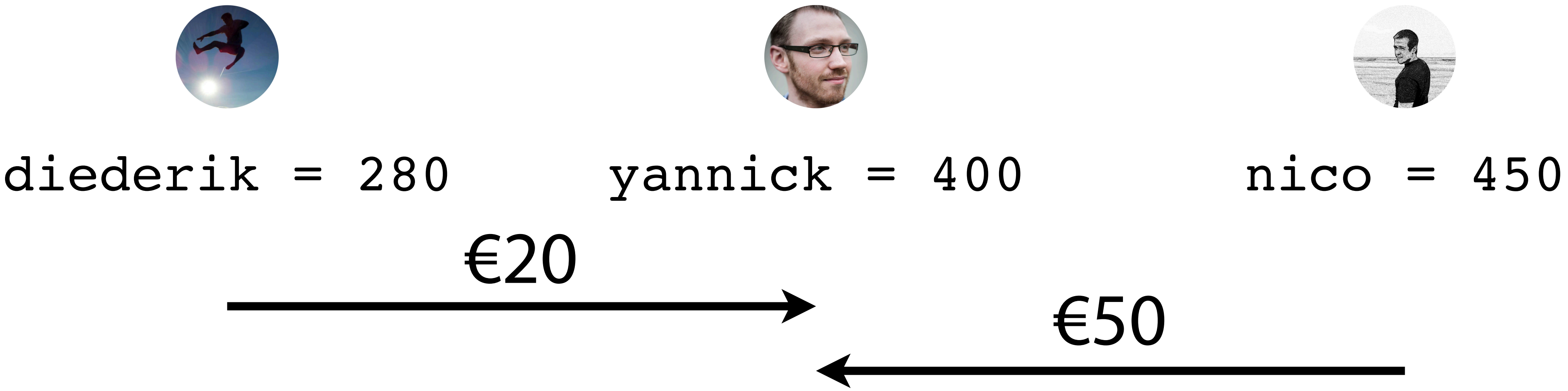
€20

€50

```
diederik = 280
yannick = 400 + 20
```

```
nico = 450
yannick = 400 + 50
```

Concurrency is difficult



```
diederik = 280      yannick = 400      nico = 450
```

€20

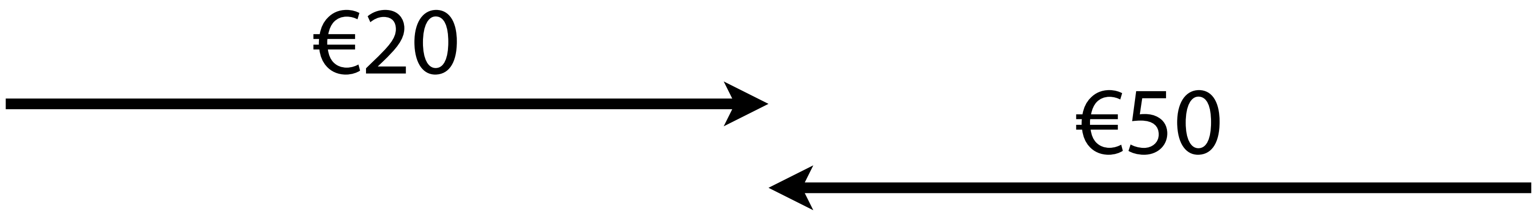
€50

```
diederik = 280
yannick = 420
```

```
nico = 450
yannick = 450
```

Concurrency is difficult

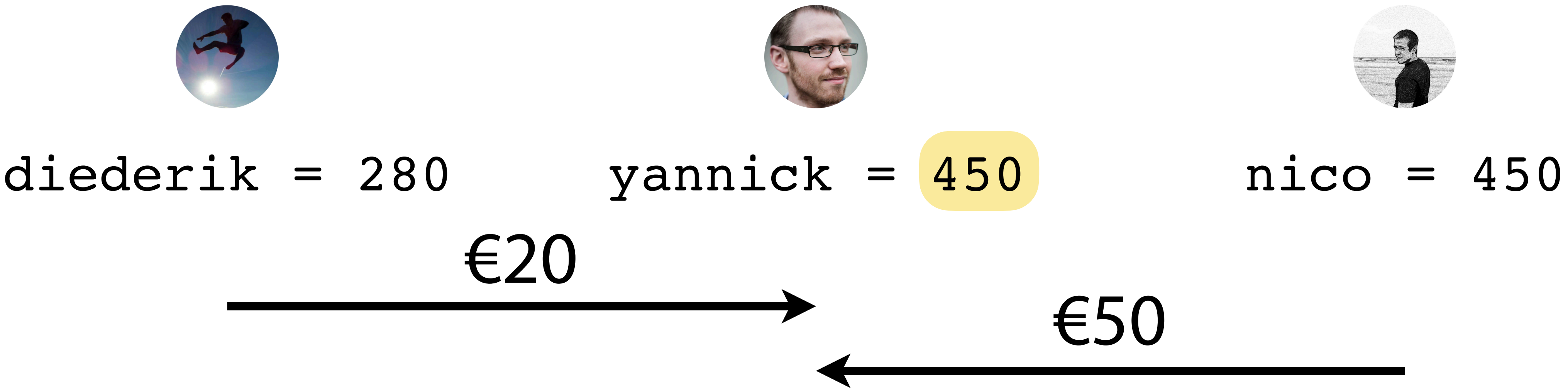
```
diederik = 280      yannick = 420      nico = 450
```



```
diederik = 280
yannick = 420
```

```
nico = 450
yannick = 450
```

Concurrency is difficult



```
diederik = 280        yannick = 450        nico = 450
```

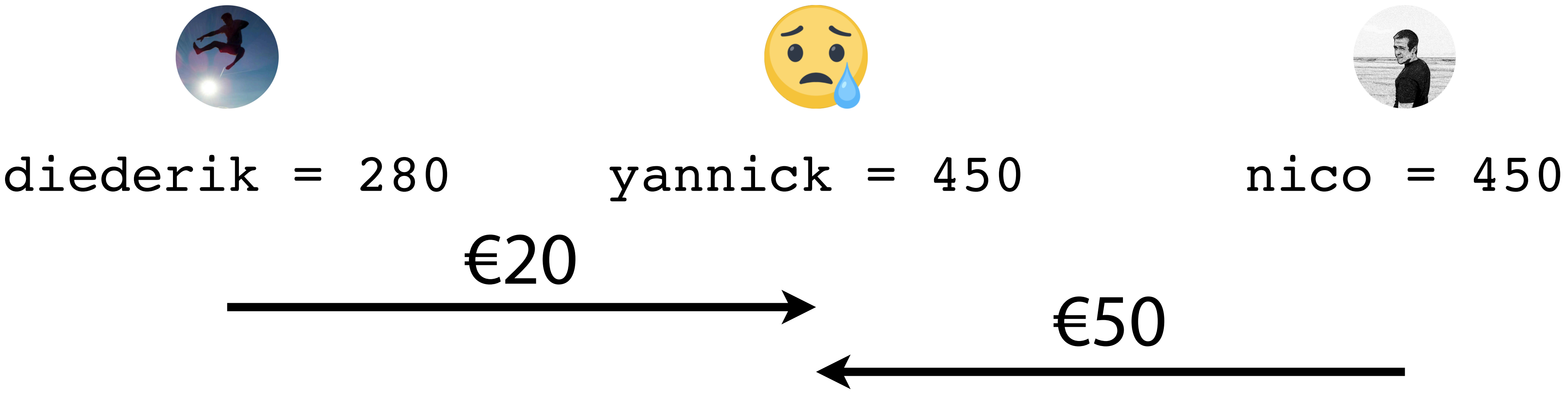
€20

€50

```
diederik = 280
yannick = 420
```

```
nico = 450
yannick = 450
```

Concurrency is difficult



```
diederik = 280        yannick = 450        nico = 450
```

```
diederik = 280
yannick = 420
```

```
nico = 450
yannick = 450
```

race condition

Concurrency is necessary but difficult

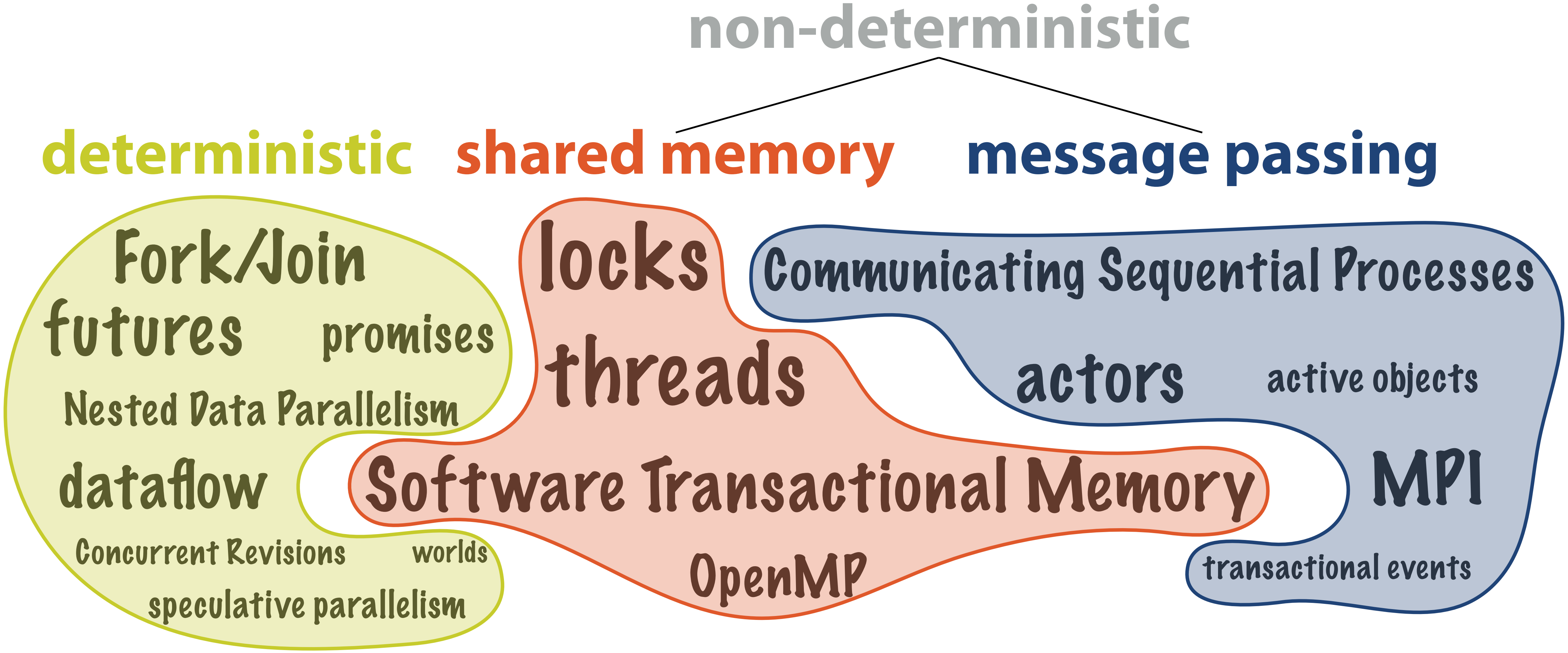
Concurrency bugs are **frequent**,
difficult to reproduce, and **difficult to debug**

⇓

Concurrency models:
set of **programming language constructs**
that introduce concurrency
but with restrictions to prevent bugs

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*

There are many different concurrency models



non-deterministic

deterministic

shared memory

message passing

Fork/Join

futures

promises

Nested Data Parallelism

dataflow

Concurrent Revisions

worlds

speculative parallelism

locks

threads

Software Transactional Memory

OpenMP

Communicating Sequential Processes

actors

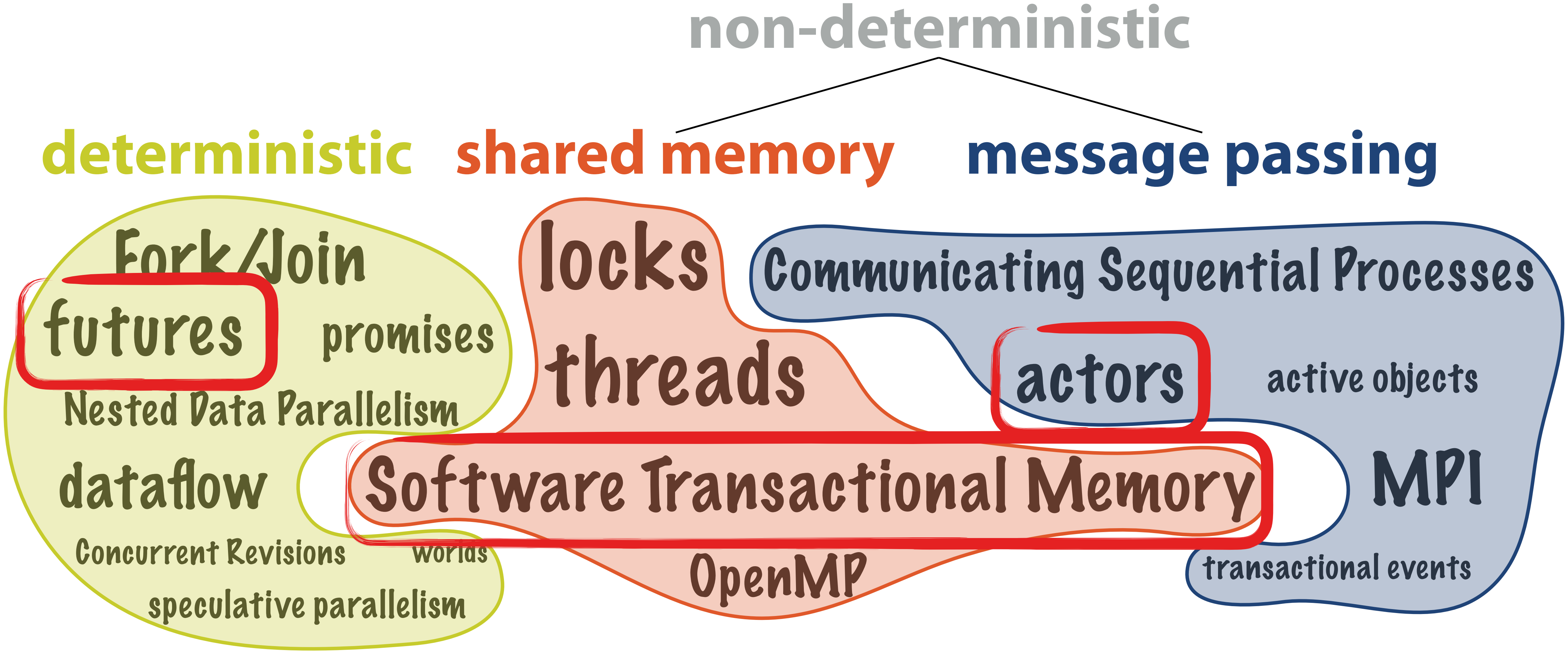
active objects

MPI

transactional events

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

There are many different concurrency models



non-deterministic

deterministic

shared memory

message passing

Fork/Join

futures

promises

Nested Data Parallelism

dataflow

Concurrent Revisions

worlds

speculative parallelism

locks

threads

Software Transactional Memory

OpenMP

Communicating Sequential Processes

actors

active objects

MPI

transactional events

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

Futures

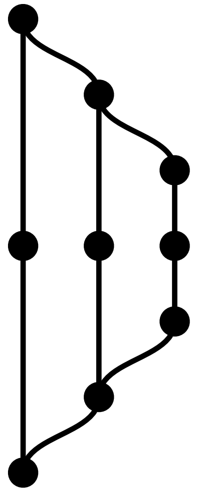
```
(def thumbnail1 (resize-image “1.jpg”))
(def thumbnail2 (resize-image “2.jpg”))
(show thumbnail1 thumbnail2)
```

Futures

```
(def thumbnail1 (fork (resize-image “1.jpg”)))
(def thumbnail2 (fork (resize-image “2.jpg”)))
(show (join thumbnail1) (join thumbnail2))
```

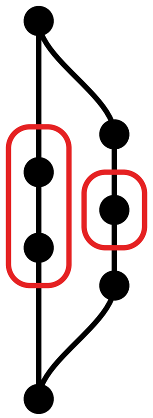
Guarantee:

Det **determinacy**



Transactions

```
(def diederik (ref 300))
(def yannick  (ref 400))
(def nico     (ref 500))
(fork
  (atomic
    (ref-set diederik (- (deref diederik) 20))
    (ref-set yannick  (+ (deref yannick)  20))))
(fork
  (atomic
    (ref-set nico     (- (deref nico)    50))
    (ref-set yannick  (+ (deref yannick) 50))))
```



Guarantees:

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

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

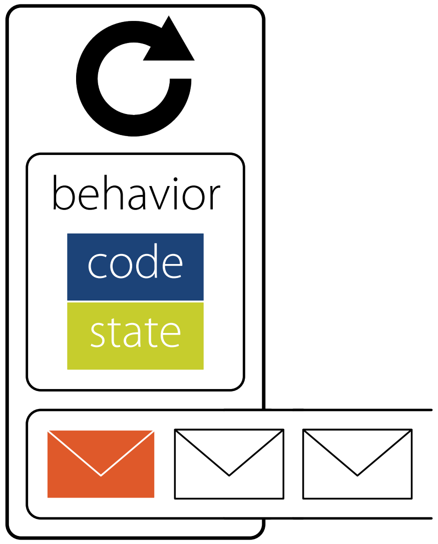
Actors

```
(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**

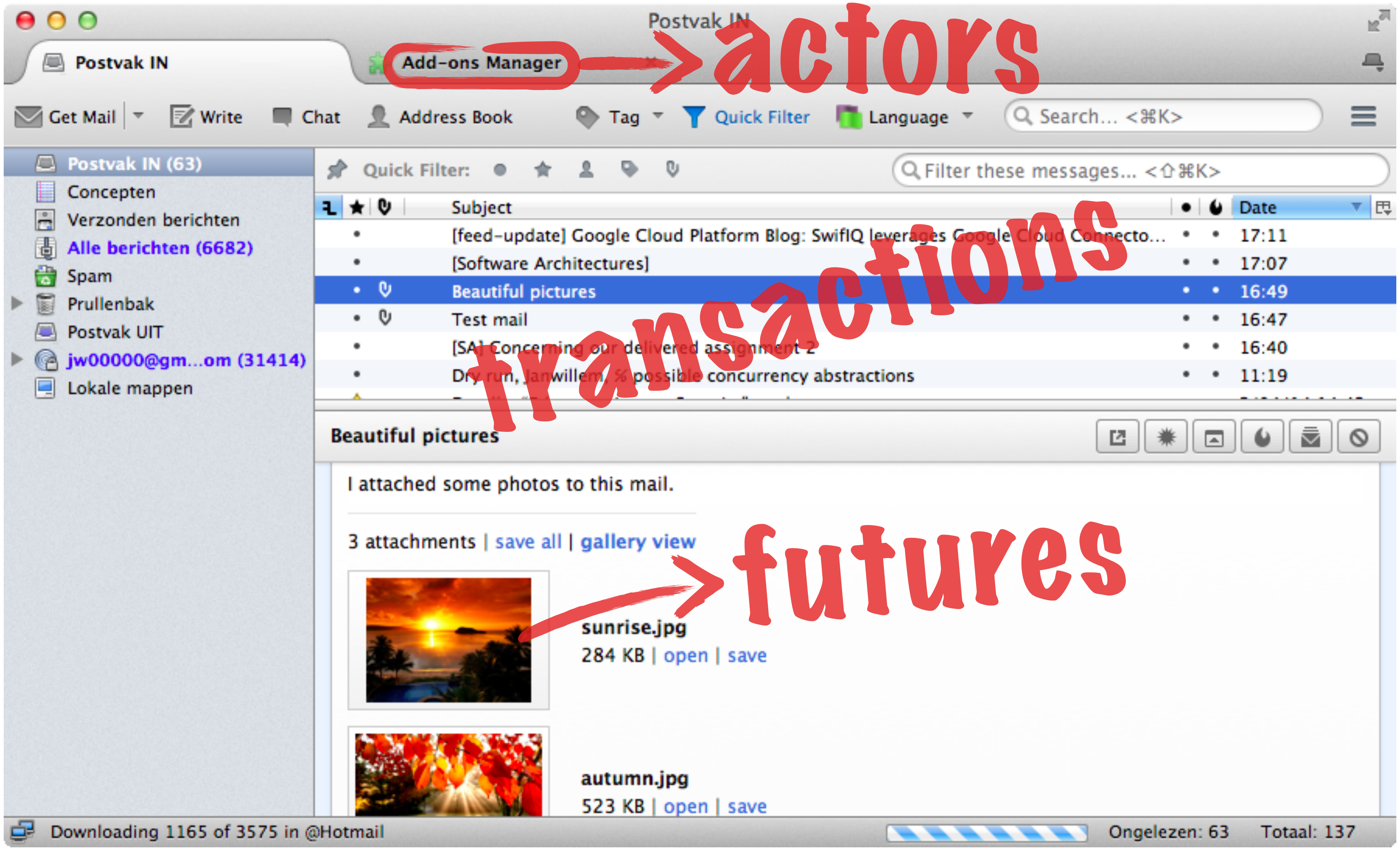


Summary

Futures	Transactions	Actors
Deterministic	*Shared memory*	*Message passing*
`(fork e)`	`(atomic e)`	`(behavior [x] [x] e)`
`(join f)`	`(ref v)`	`(spawn b v)`
	`(deref r)`	`(send a v)`
	`(ref-set r v)`	`(become b v)`
Det Determinacy	Iso Isolation	ITP Isolated turn principle
	Pro Progress	DLF Deadlock freedom

Formalization of three separate models ☞ Chapter 2

Different concurrency models target different use cases



Subject	Date
[feed-update] Google Cloud Platform Blog: SwiftIQ leverages Google Cloud Connecto…	17:11
[Software Architectures]	17:07
Beautiful pictures	16:49
Test mail	16:47
[SA] Concerning our delivered assignment 2	16:40
Dry run, Janwillem, % possible concurrency abstractions	11:19

Beautiful pictures

I attached some photos to this mail.

3 attachments | save all | gallery view

sunrise.jpg
284 KB | open | save

autumn.jpg
523 KB | open | save

Downloading 1165 of 3575 in @Hotmail

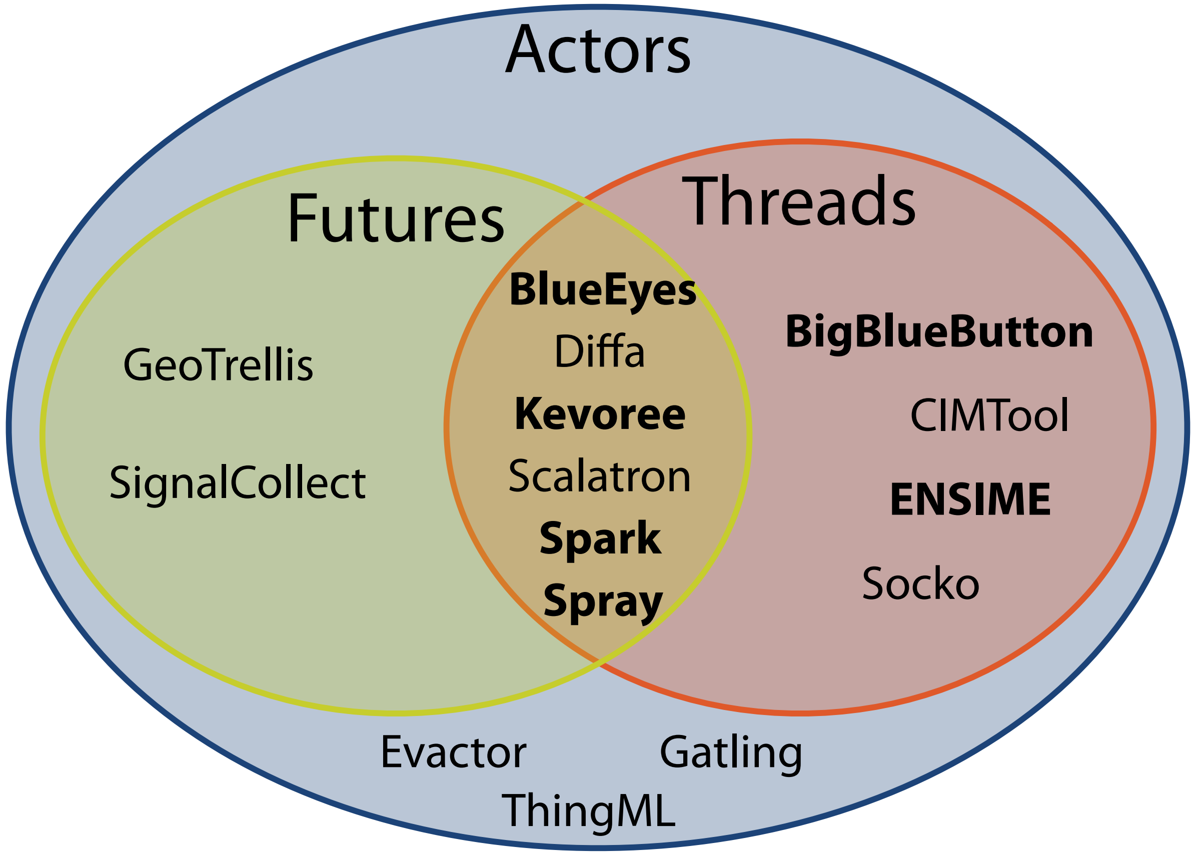
Ongelezen: 63 Totaal: 137

actors

transactions

futures

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)

Developers **combine** concurrency models.

Programming languages allow this.

How does this affect their **guarantees**?

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 …))
```

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

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

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

Naive combinations lead to problems

3 common problems:

- spurious retries

  ⇒ races

- unexpected blocking

  ⇒ deadlocks

- unexpected retries

  ⇒ livelocks

Caused by operations that:

- retry
- block

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

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

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

Naive combinations lead to problems

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

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

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

		inner					
	Livelocks	Atom	Agent	STM	Future	Promise	Channel
outer	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

Unified model of futures, transactions, and actors that:

1 Separate models: backward compatibility

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

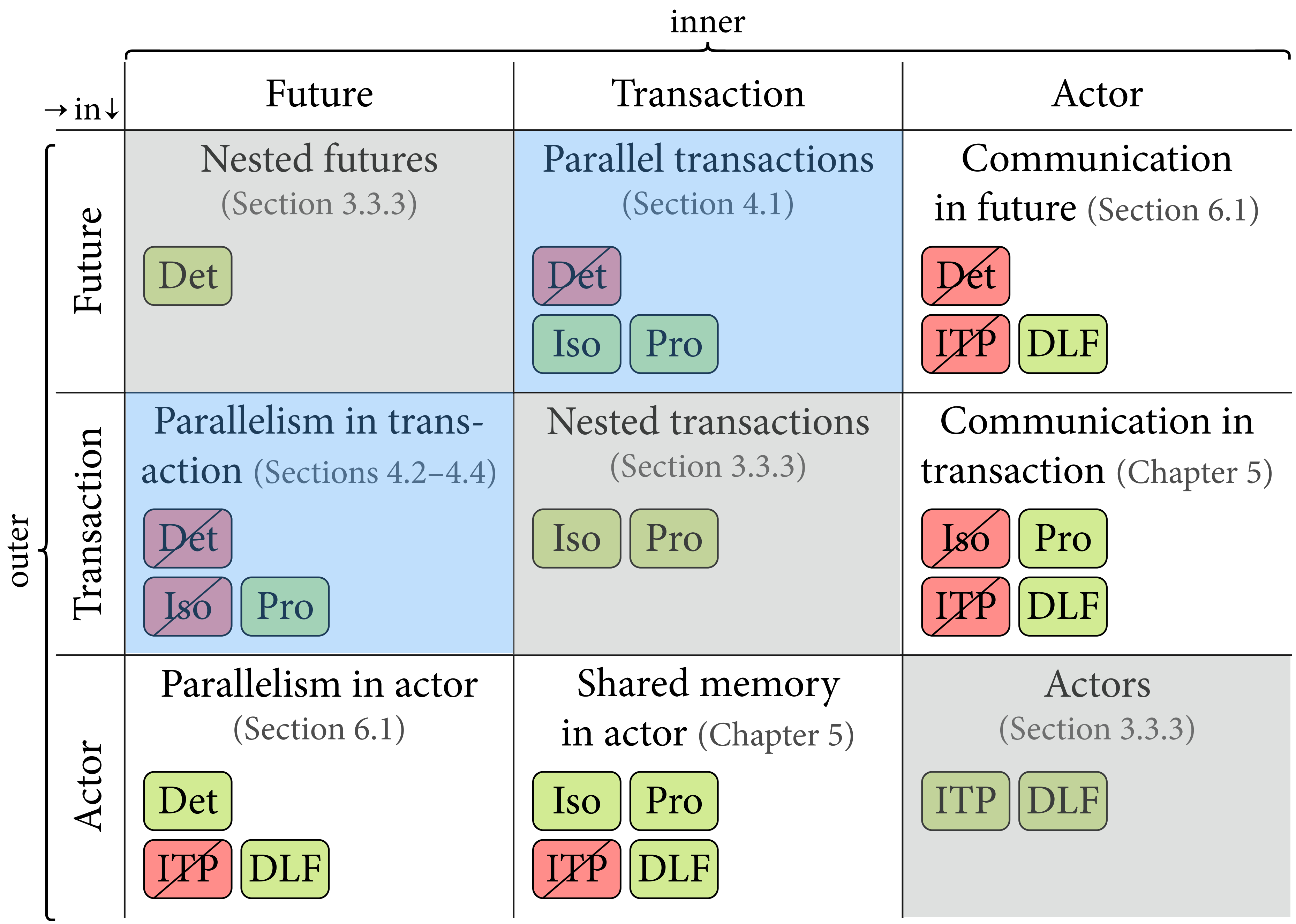
“Naive” combinations

		inner		
outer	→ 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~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
	Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

Trivial combinations

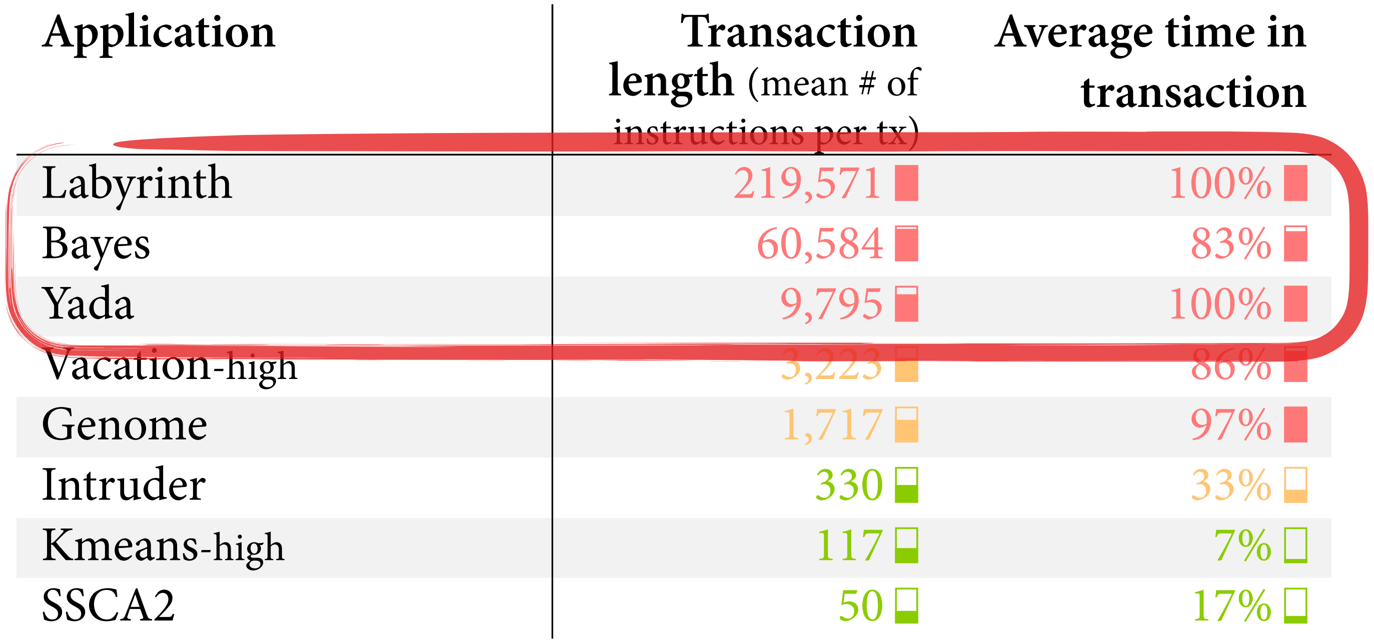
		inner		
	→ in ↓	Future	Transaction	Actor
outer	Future	Nested futures (Section 3.3.3) Det	Parallel transactions (Section 4.1) ~~Det~~ Iso Pro	Communication in future (Section 6.1) ~~Det~~ ~~ITP~~ DLF
	Transaction	Parallelism in transaction (Sections 4.2–4.4) ~~Det~~ ~~Iso~~ Pro	Nested transactions (Section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

Transactions + Futures



→ 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~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

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)

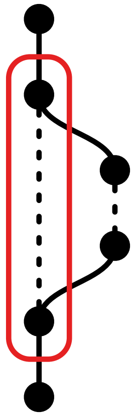
Problems when creating future in transaction

Impure languages (e.g. Clojure, ScalaSTM)

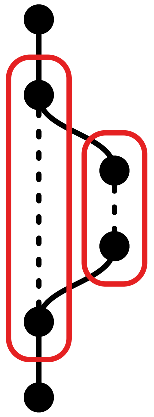
Tasks in transaction do not share context

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

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



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

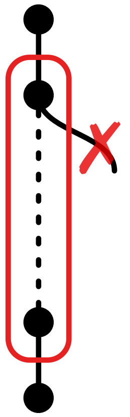


Pure languages (Haskell)

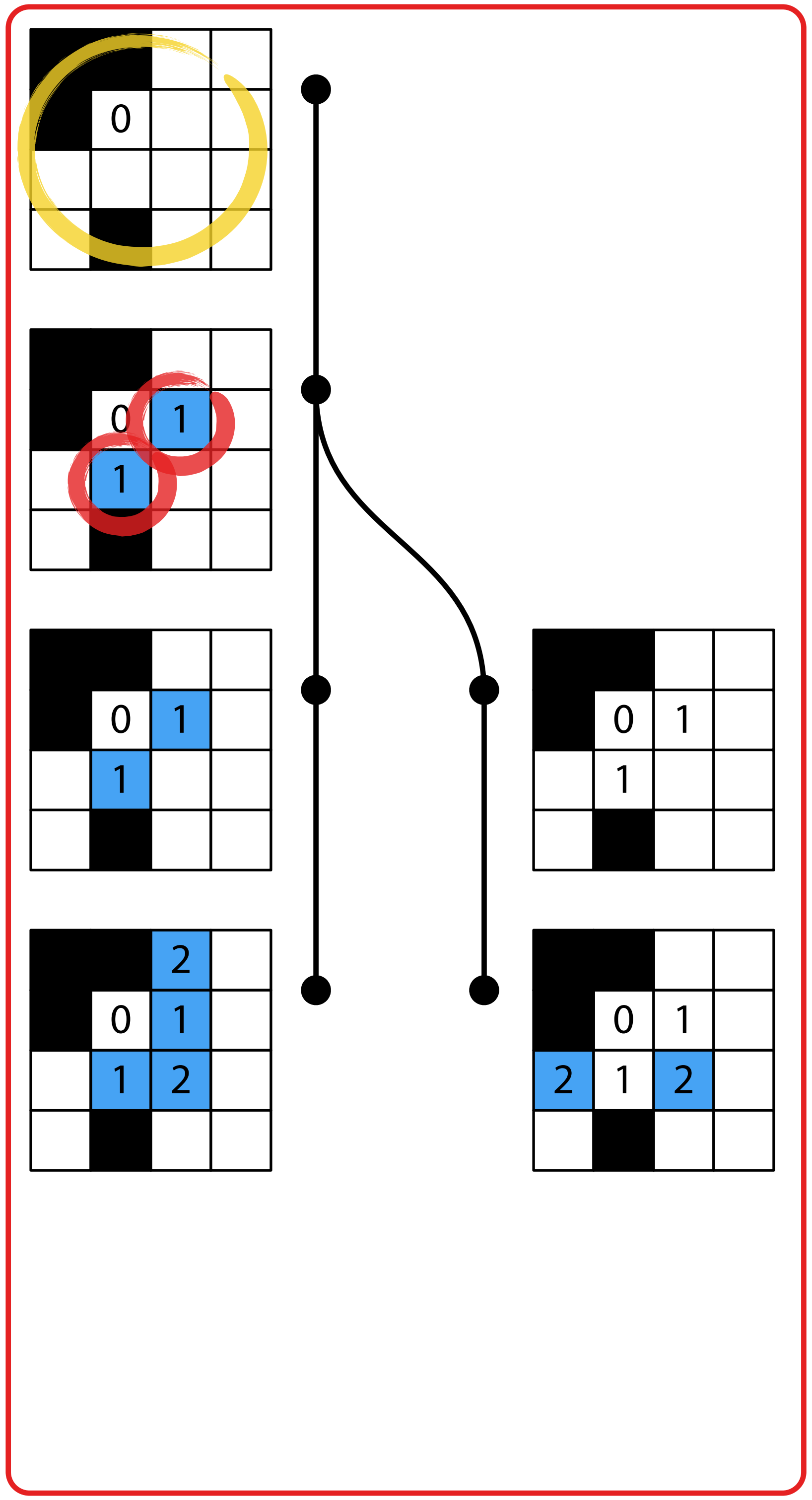
Tasks in transaction prohibited

⇒ isolation guaranteed but **parallelism limited**

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

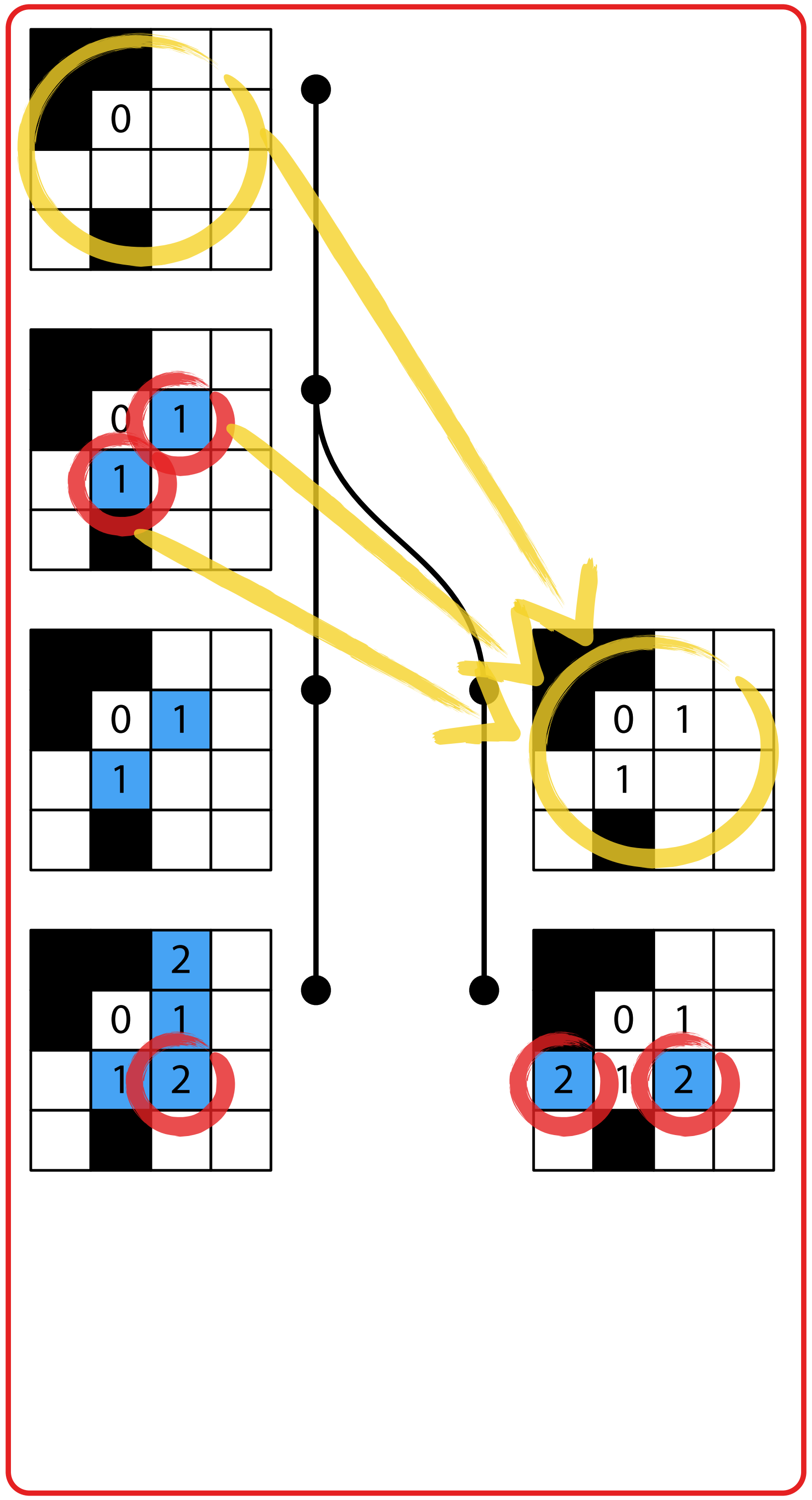


Transactional Futures



```
(atomic
   (ref-set … 1)
```

`fork` creates isolated task



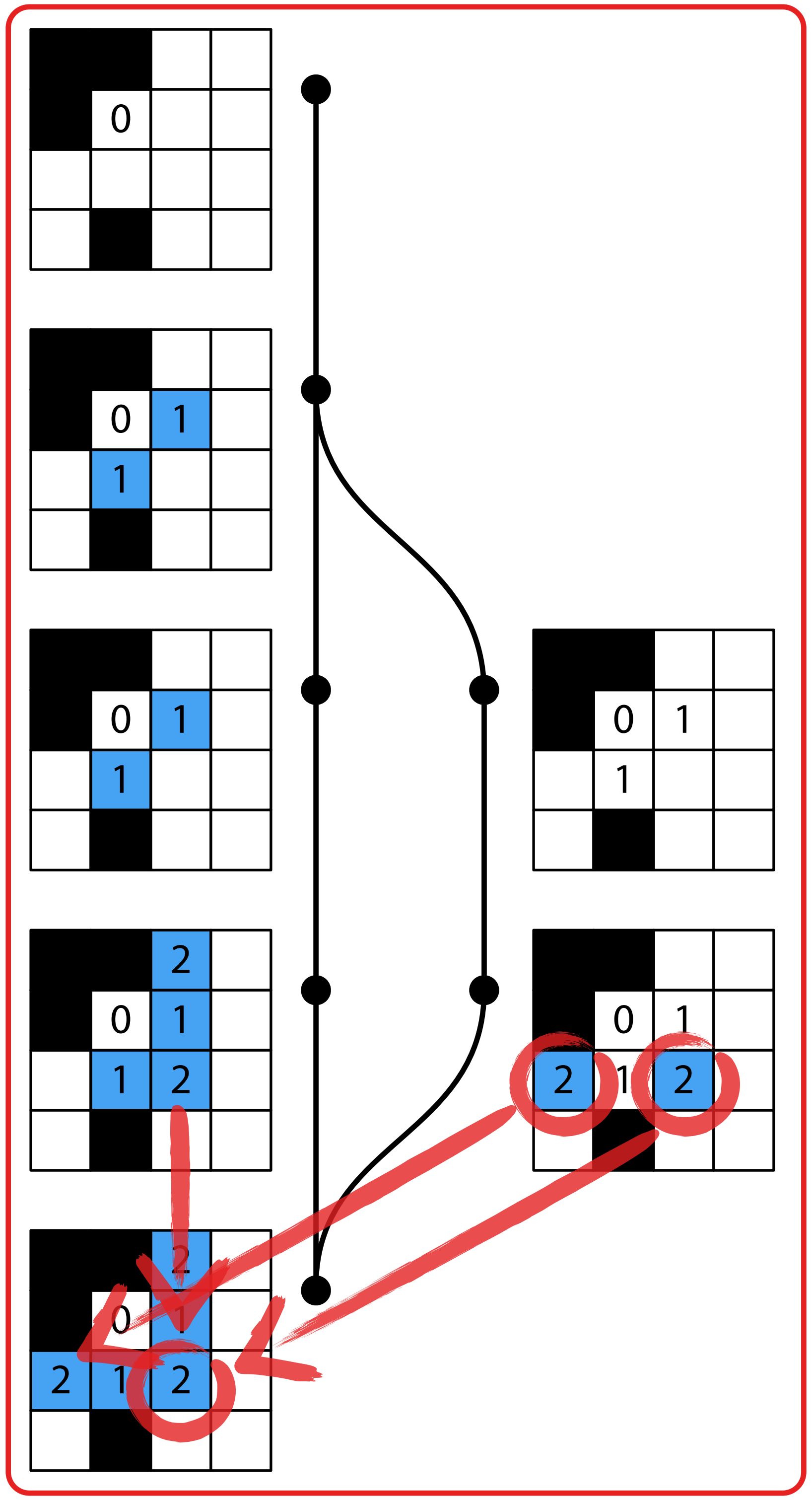
```
(atomic
  (ref-set … 1)
  (fork
    (ref-set … 2))
  (ref-set … 2)
```

Each transactional task contains:

snapshot: transactional state on creation

local store: local modifications

`join` merges changes

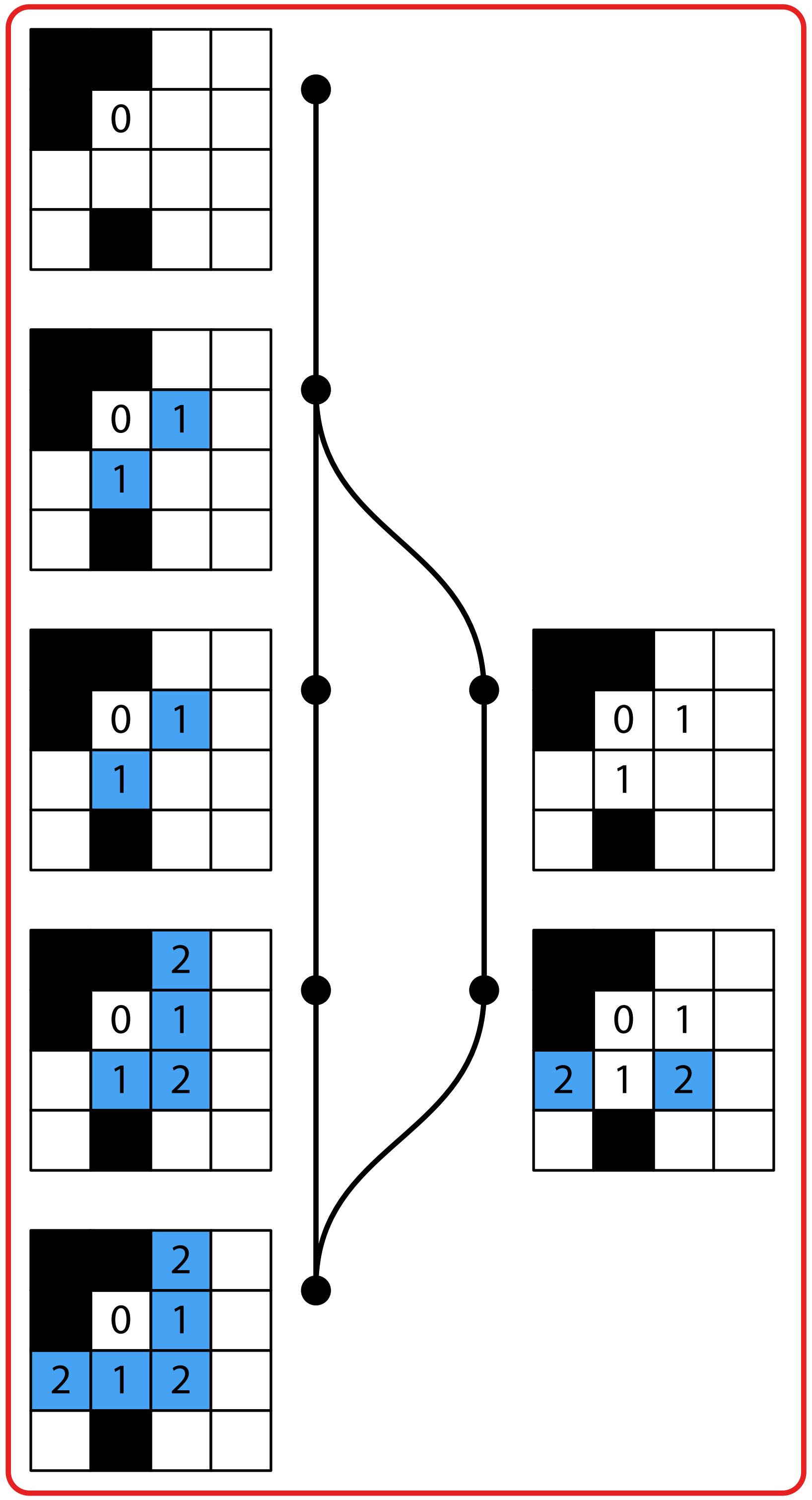


```
(atomic
   …
   (join child))
```

merge local store of child into parent

Conflict resolution function: `(ref 0 resolve)`

All tasks commit atomically
⇒ isolation and progress maintained



```
(atomic
  …
  (join child))
```

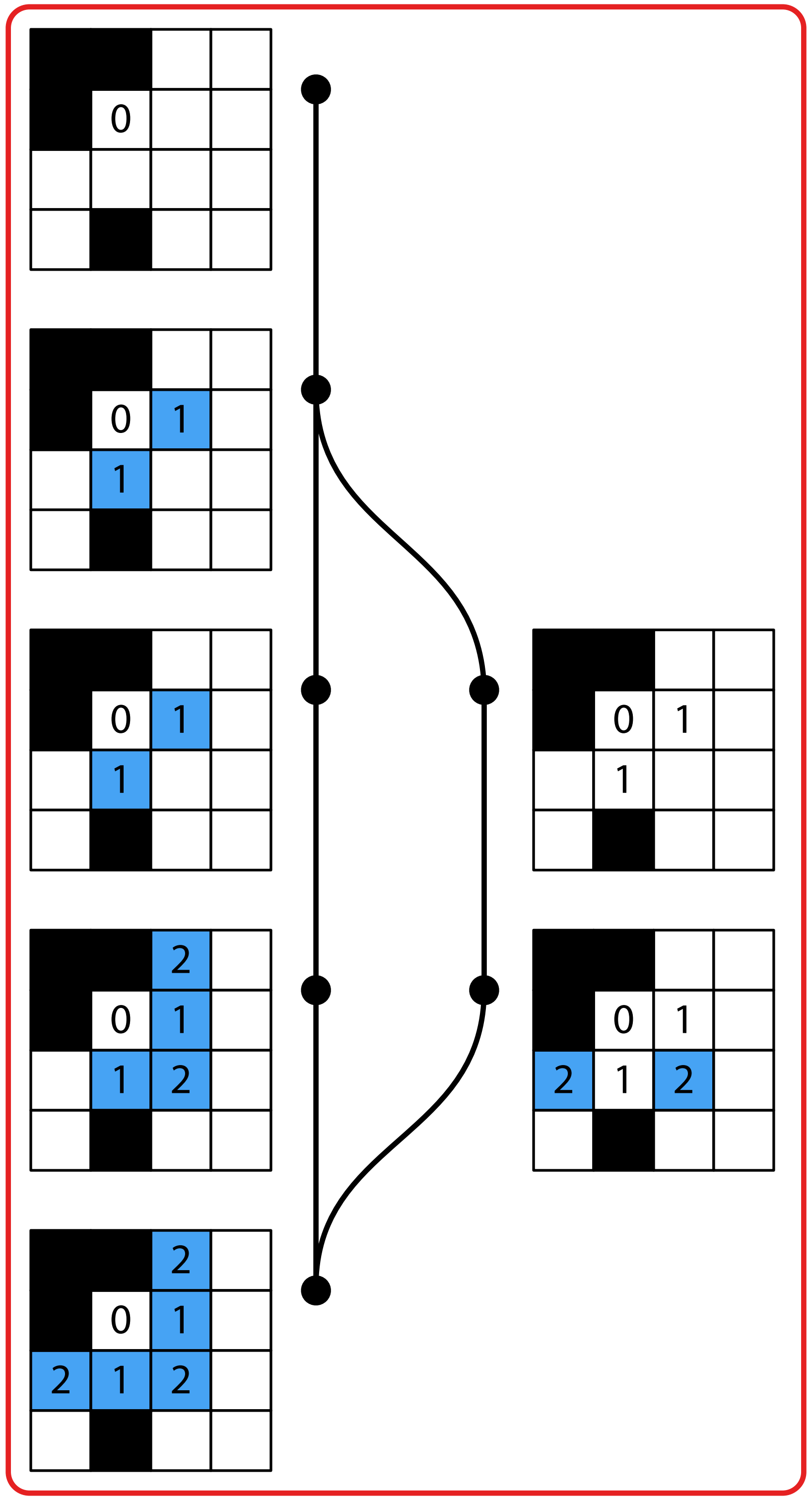
All tasks must be joined before commit

⇒ **isolation maintained** Iso

progress maintained Pro

~~Det~~ → ITD

Intratransaction determinacy

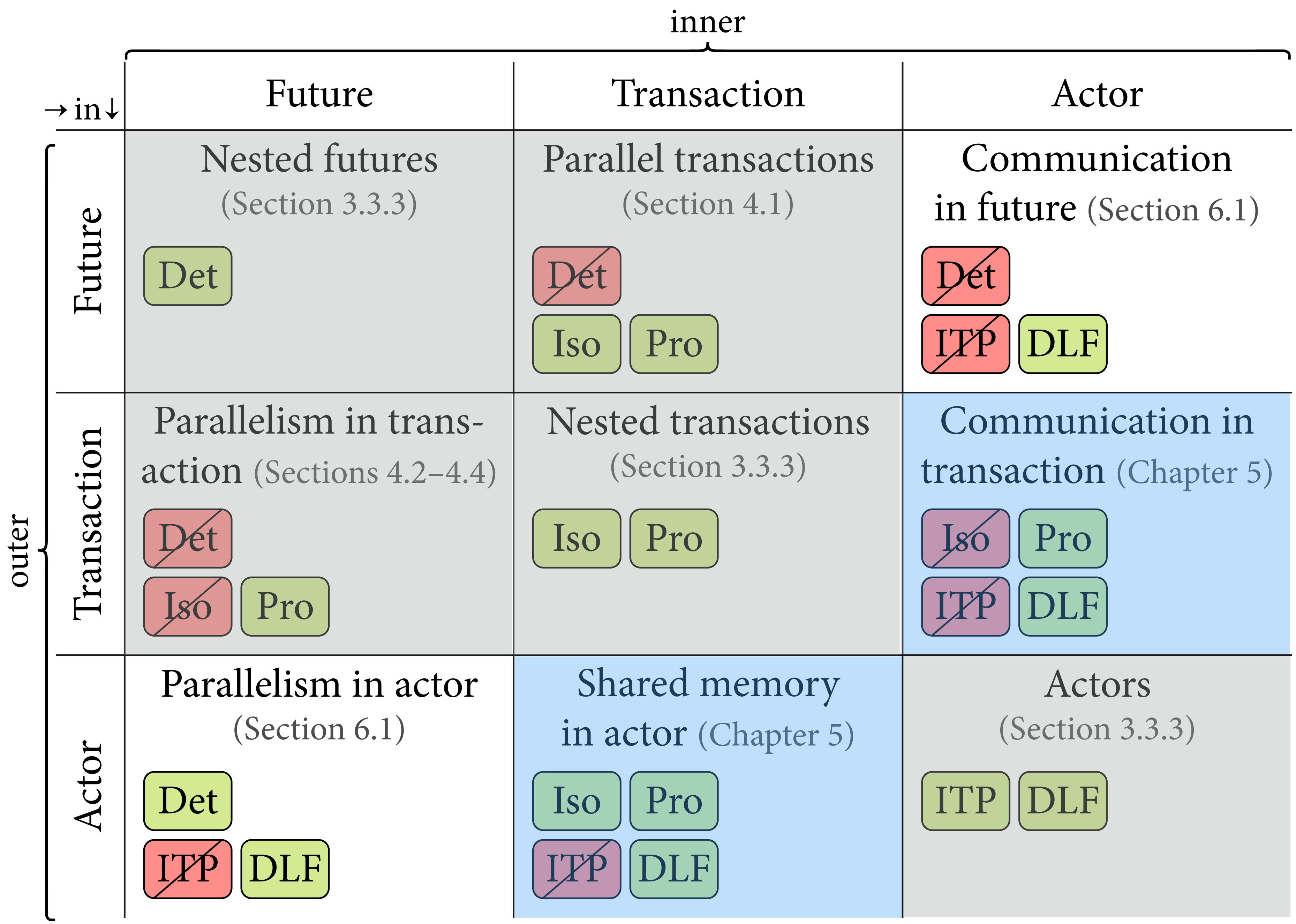


Transactions can commit in any order
⇒ ~~Det~~ inevitable

But: determinacy *within* each transaction
= **intratransaction determinacy** ITD

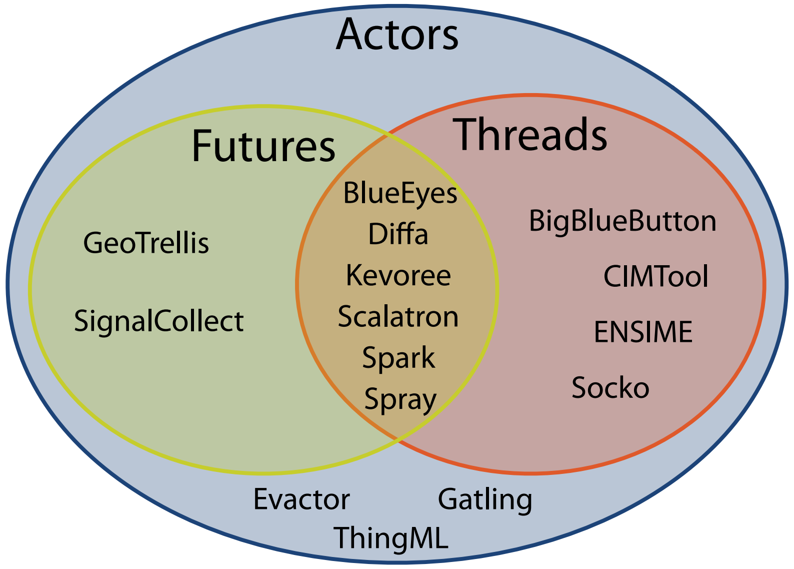
And isolation *between* transactions Iso

Transactions + Actors



		inner		
	→ in ↓	Future	Transaction	Actor
outer	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~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
	Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

Motivation: (1) safe shared information between actors

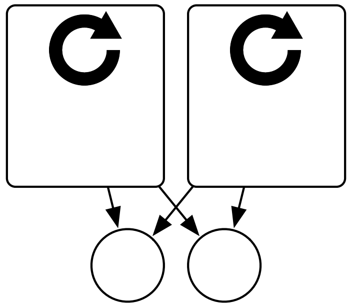


Impure actor languages (e.g. Scala)

10/15 projects introduce shared memory

⇒ **ITP** **broken** ~~ITP~~

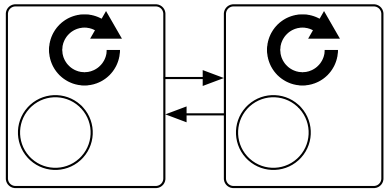
⇒ **races & deadlocks** **possible**

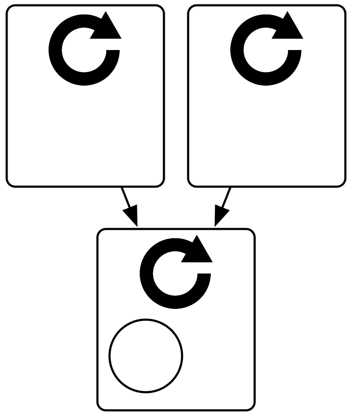


Pure actor languages (e.g. Erlang)

Patterns: replication/delegation

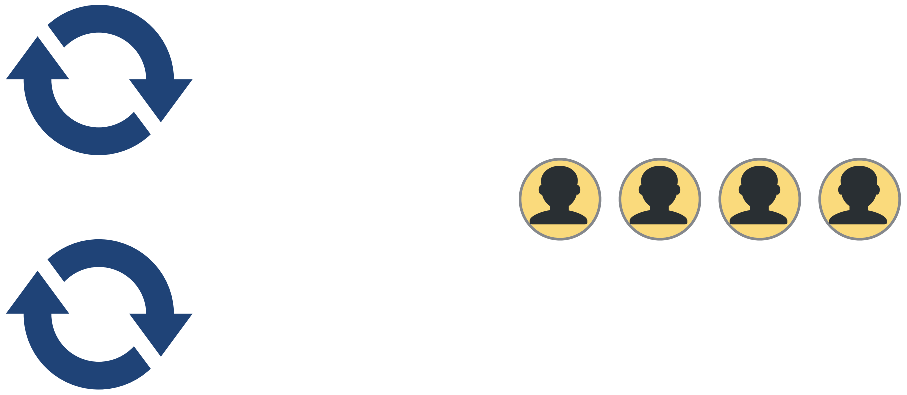
⇒ ITP guaranteed but
safety **up to the developer**





Motivation: (2) communication between transactions

“Vacation” processes customers in parallel

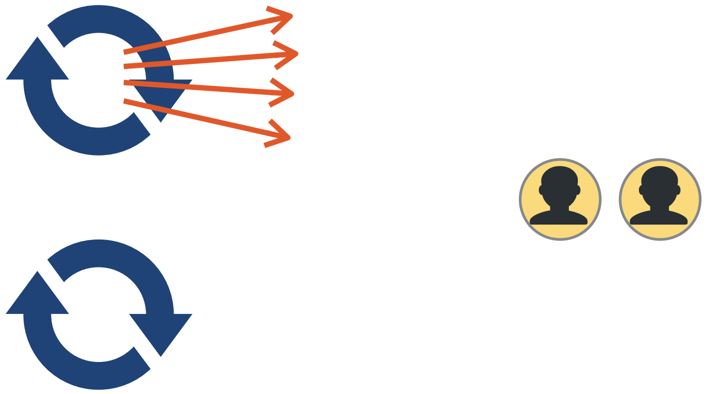


```
(def customer-behavior
  (behavior [id] [c]
    (atomic
      (reserve-flight (:orig @c) (:dest @c) (:start @c))
      (reserve-flight (:dest @c) (:orig @c) (:end @c))
      (reserve-room   (:dest @c) (:start @c) (:end @c))
      (reserve-car    (:dest @c) (:start @c) (:end @c))
      (ref-set c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

separate from transaction, no side-effect ✓

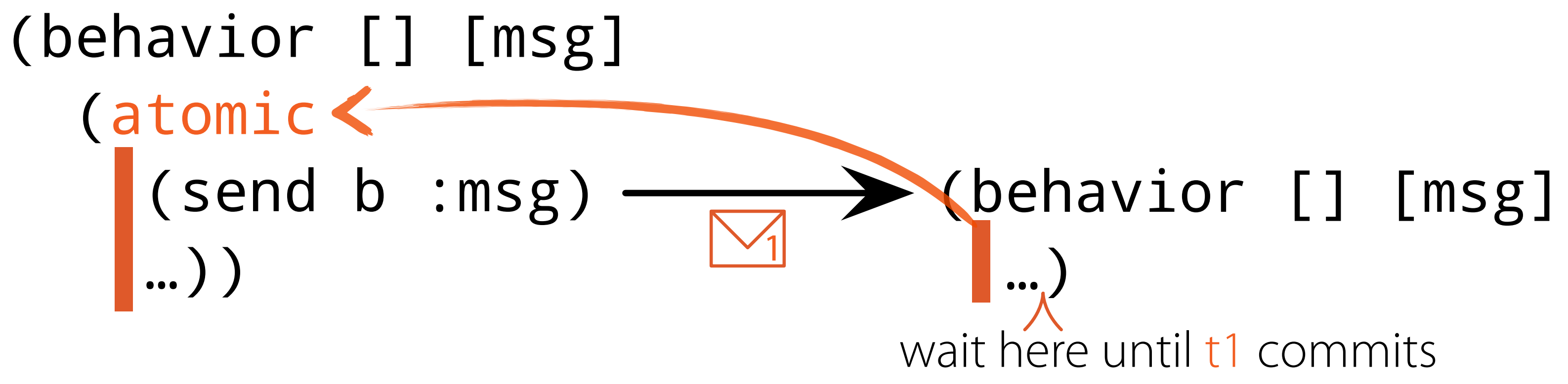
```
  (spawn airline-beh flights)
  (become airline-beh flights)
```

delay side effect until commit (pessimistic) ↷

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

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

Sending a message in a transaction



```
(behavior [] [msg]
  (atomic
    (send b :msg)
    …))
```

```
(behavior [] [msg]
  …)
```

wait here until t1 commits

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

		inner		
	→ in↓	Future	Transaction	Actor
outer	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~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
	Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

Chocola:
c^{h}omposable concurrency language

inner

	→ in ↓	Future	Transaction	Actor
outer	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

Implementation

Extension of Clojure

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

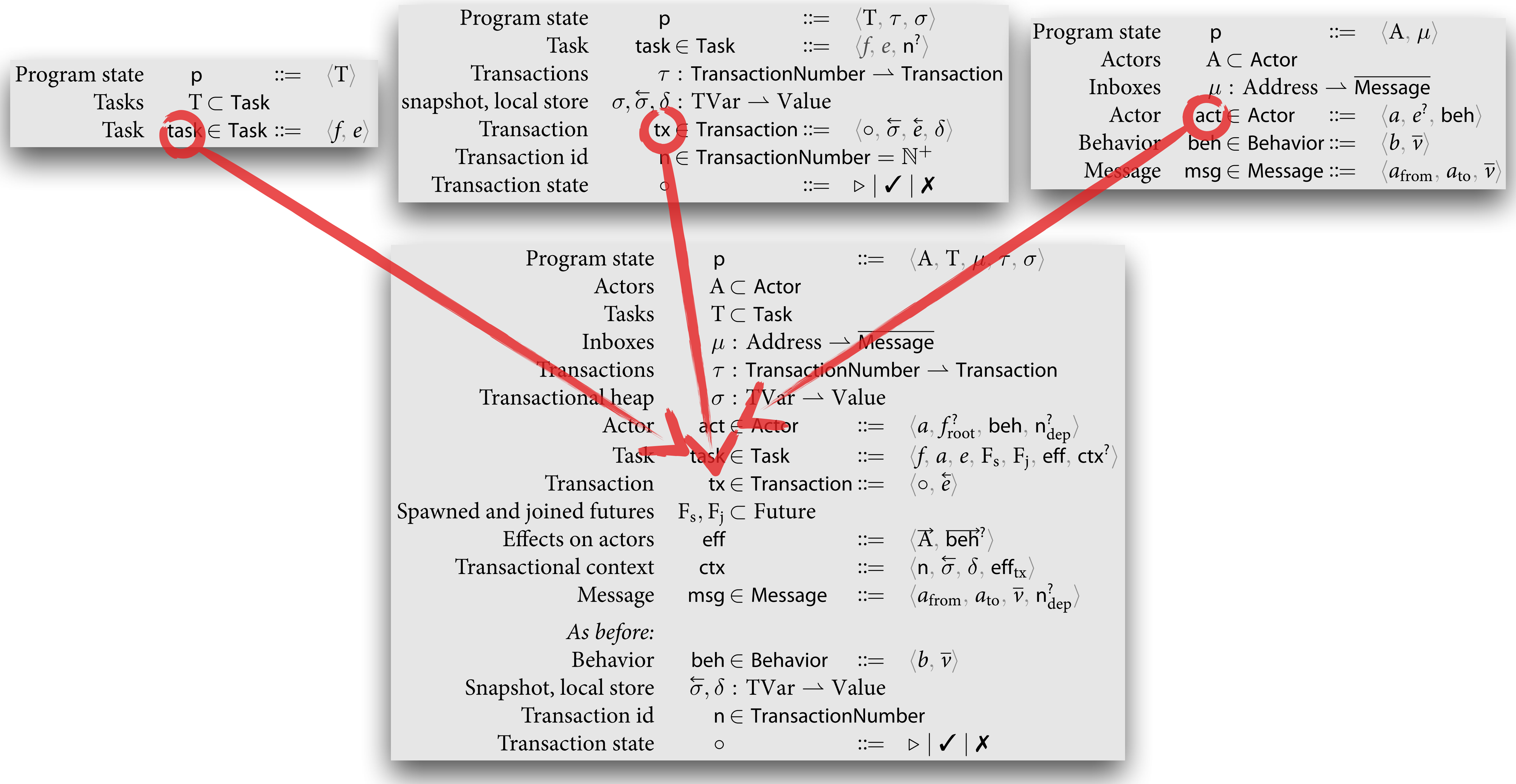
} added

☞ 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	::=	$\langle T \rangle$
Tasks	$T \subset$ Task		
Task	task $\in$ Task	::=	$\langle f, e \rangle$

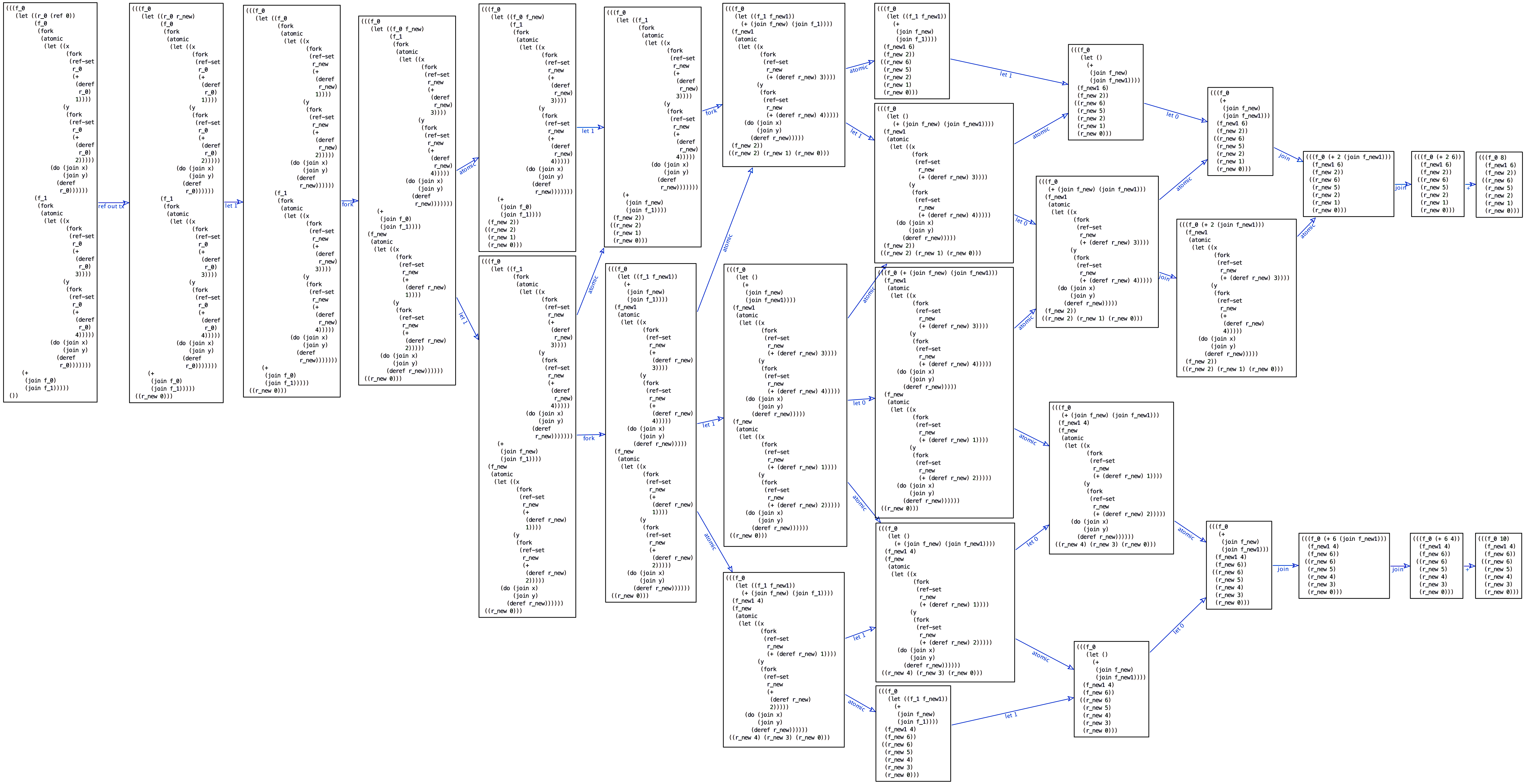
Program state	p	::=	$\langle T, \tau, \sigma \rangle$
Task	task $\in$ Task	::=	$\langle f, e, \mathsf{n}^? \rangle$
Transactions	τ : TransactionNumber $\rightharpoonup$ Transaction		
snapshot, local store	$\sigma, \overleftarrow{\sigma}, \delta$: TVar $\rightharpoonup$ Value		
Transaction	tx $\in$ Transaction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

Program state	p	::=	$\langle A, \mu \rangle$
Actors	$A \subset$ Actor		
Inboxes	μ : Address $\rightharpoonup$ $\overline{\text{Message}}$		
Actor	act $\in$ Actor	::=	$\langle a, e^?, \mathsf{beh} \rangle$
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	::=	$\langle A, T, \mu, \tau, \sigma \rangle$
Actors	$A \subset$ Actor		
Tasks	$T \subset$ Task		
Inboxes	μ : Address $\rightharpoonup$ $\overline{\text{Message}}$		
Transactions	τ : TransactionNumber $\rightharpoonup$ Transaction		
Transactional heap	σ : TVar $\rightharpoonup$ Value		
Actor	act $\in$ Actor	::=	$\langle a, f^?_{\text{root}}, \mathsf{beh}, \mathsf{n}^?_{\text{dep}} \rangle$
Task	task $\in$ Task	::=	$\langle f, a, e, F_s, F_j, \mathsf{eff}, \mathsf{ctx}^? \rangle$
Transaction	tx $\in$ Transaction	::=	$\langle \circ, \overleftarrow{e} \rangle$
Spawned and joined futures	$F_s, F_j \subset$ Future		
Effects on actors	eff	::=	$\langle \overrightarrow{A}, \overrightarrow{\mathsf{beh}}^? \rangle$
Transactional context	ctx	::=	$\langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}} \rangle$
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	$\circ$	::=	$\triangleright \mid ✓ \mid ✗$

Formalization of Chocola ☞ Chapter 7

Executable formal semantics with PLT Redex



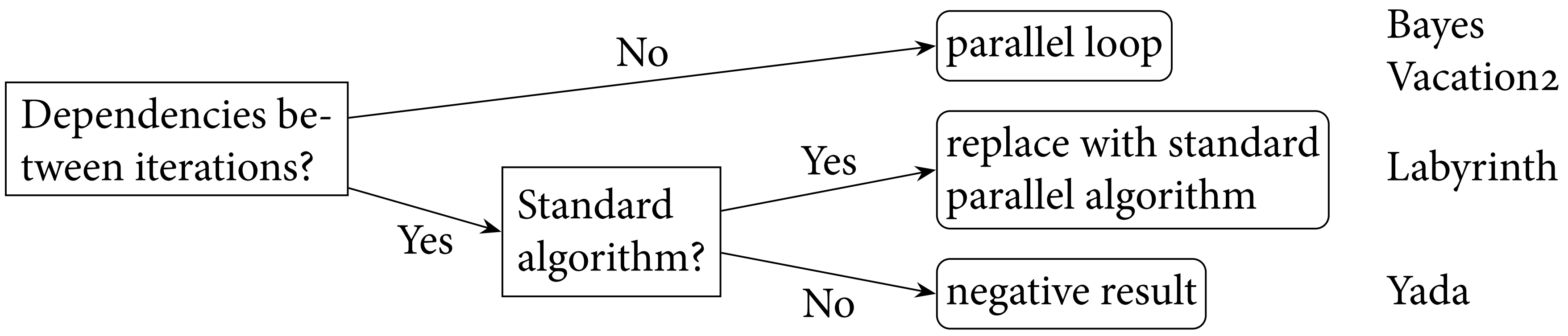
☞ https://github.com/jswalens/chocola-redex

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



Dependencies between iterations?

No → parallel loop: Bayes, Vacation2

Yes → Standard algorithm?

Yes → replace with standard parallel algorithm: Labyrinth

No → negative result: Yada

③ evaluation criteria

performance: speed-up

developer effort: lines changed + qualitative assessment

Summary of results

	Speed-up original		Speed-up Chocola	Lines of code added	
Labyrinth	1.3	↗	2.3	+11%	8 cores
Bayes	2.8	↗	3.5	+1	
Vacation2	2.6	↗	33.2	+8%	64 cores
Yada	futures/actors not applicable				

Better performance for little effort

☞ Chapter 8

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

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; accepted]
  - 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* (accepted for AGERE'18)

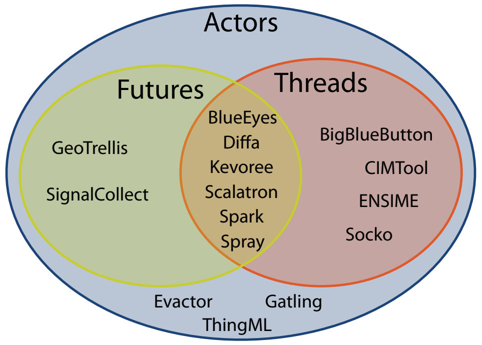
Future work

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

Conclusion

Concurrency models are combined

Naive combinations violate guarantees



		inner					
	Races	Atom	Agent	STM	Future	Promise	Channel
outer	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